An Object Oriented Design for the Visualization of Multi-Variable Data Objects

Jean M. Favre and James Hahn

EE&CS Department
The George Washington University
Washington, DC 20052

Abstract

This paper presents an object-oriented system design supporting the composition of scientific data visualization techniques based on the definition of hierarchies of typed data objects and tools. Traditional visualization systems focus on creating graphical objects which often cannot be re-used for further processing. Our approach provides objects of different topological dimension to offer a natural way of describing the results of visualization mappings. Serial composition of data extraction tools is allowed, while each intermediate visualization object shares a common description and behavior. Visualization objects can be re-used, facilitating the data exploration process by expanding the available analysis and correlation functions provided. This design offers an open-ended architecture for the development of new visualization techniques. It promotes data and software re-use, eliminates the need for writing special purpose software and reduces processing requirements during interactive visualization sessions.

1. Introduction

1.1. Visualization Objects and Processing Modules

We examined several well-known visualization systems (AVS, Explorer, FAST, Khoros, ...) and the techniques they use to interconnect independent data processing modules. A typing system is usually provided to define process interfaces, and connections are only allowed between I/O ports accepting the same *type* of data. For this paper, the word *type* is used in the programming language sense for input and output abstract data structures. Graphical output is usually the end-product of a visualization session. Thus, data of various types are passed between modules - each module possibly creating data of a new type - until displayable geometry is output for a final rendering. Data-flow architectures already support some form of data sharing. However, the data produced for the graphical display stage of a visualization are collections of graphics primitives, using a system-defined geometry type. Rendering/display modules become in effect a bottleneck, taking *geometric* data in, without allowing their re-use by non-graphical processing tools. Common visualization tools have too often focused on directly producing geometric data (for example, sets of polygons or line segments) ready for the fast hardware rendering workstations of the 1990's. Thus, individual polygons or lines are generated with normal information and a color index or an RGB value associated with each vertex. Rendering such objects can often be done in real time but unfortunately, rendering is the <u>only</u> operation that can be applied to such objects.

1.2. Previous work

Foley and Lane [2] have presented multi-valued visualization techniques. They assume the definition of a geometric object D (which can be formed of several disjoint surfaces). D is a user-defined surface or an iso-surface and is used as a value probe to examine volume data at the surface of the object. Color Blended Contours, Projected Surface Graphs, Contour Curves on a Projected Surface Graph, Iso-Surface and Hyper-Surface Projection Graphs are the tools they use to compose volumetric rendering techniques. However, their work is restricted to the use of surfaces for geometric support, and their analysis of data is limited to rendering operations.

The definition and use of object-oriented abstract data *types* for scientific visualization has been documented by several authors [3,6,8,11]. However, they do not deal with composition techniques and a system design to support these techniques. The data types provided do not foster the re-use of data objects for composition purposes. In a Visualization '91 Workshop Report [1], the notion of "Functions of Several Variables" (FOSV) is discussed as the most important abstract data type relevant to scientific computations. The workshop participants propose to use this data class as the starting point for a reference model, and consider Visualization mappings as operations between FOSV instances. Lucas et al. [8] use this paradigm and present a high-level overview of Data Explorer. Our system design differs from their approach by putting more emphasis on data integrity and requirements for Functional Composition.

1.3. Motivation and Design Goals

To promote serial composition of visualization techniques, we must augment the inter-connectivity of modules with carefully designed *typed* output and allow several visualization primitives to form a composite visualization object. We give visualization objects a broader functionality than pure graphics primitives. An object-oriented design is used to encapsulate both geometry and field data so that objects can be freely exchanged between data operators. Each data class is endowed with data extraction and rendering operators. Our emphasis is on combining visualization techniques to create composite visualization designs and providing an abstraction which favors data and code re-use. It is particularly useful for multi-variate field data, where more than one scalar field is considered at each grid point, and where we can alternate between different data field views, independently of the underlying geometry.

In Section 2, we briefly introduce the object-oriented paradigm. Section 3 shows how most sets of scientific data can be described by the generic unstructured data grids in 3-D space and gives details about the general purpose sets of line-, surface- and volume-elements of points and the functions associated with them. Section 4 shows how data visualization tools are re-defined and combined to achieve Data Selection and Functional Composition. Section 5 describes several examples. We offer some discussion in section 6 and conclude in section 7.

2. The Object-Oriented Paradigm

In most conventional programming languages, every name (identifier) has a type associated with it. This type determines what operations can be applied to the name. For example, integer and floating point *types* come with the pre-defined +, -, *, / operators. The programming paradigm provided by object-oriented languages favors a similar process of defining types and associated operators.

Abstraction is defined as the process of extracting essential properties of a concept. Data structures allow the abstraction of the structural aspects of the data organization. Procedures and functions allow the abstraction of behavioral aspects. The C++ programmer can combine these two user-defined abstractions to create *data classes*, defining new types for which access to data is restricted to a specific set of access functions. The data structure thus embedded in a class definition can be initialized, accessed and operated upon by ways of these access functions. These functions are shared by all instances of the class and common behavior among them is thus insured.

Sub-classes can be derived from a parent class by sharing data structures and operations. Inheritance is the technique which allows sub-classes of a parent class to re-use (inherit) the parent's functions. Using the object-oriented paradigm, our aim is to define high-level classes for each set of data common to the visualization field. Associated with these definitions are display and processing tools needed to operate on such sets of data.

3. Data Classes based on the Spatial Domain

Gridded data are a common occurrence in scientific computations. Data may come in regular, rectilinear, curvilinear or arbitrary grids. As Butler et al. have remarked [1], visualization mappings generally use and produce sets of points in space, with associated data values. Furthermore, common data extraction tools are generally mappings from n-D space to (n-k)-D space and most sets of data can be described by generic data structures in such spaces.

To manipulate sets of points, several user-defined types are used. We use a **Point** class for points in R^3, with associated data values. The elementary linear segment joining two **Points** is defined by the class **LineCell**. Likewise, the class **SurfaceCell** is represented by sub-classes of the basic surface elements (triangle, quad, etc.), and the class **VolumeCell**, with sub-classes of elementary volume elements (tetrahedron, prism, hexahedron, etc.). The C++ classes **PointSet**, **LineSet**, **SurfaceSet** and **VolumeSet** are then used to organize *sets of* elementary objects of the respective types. A class hierarchy exists to combine cells of identical topological dimension, and the sets only manipulate pointers to the 1-D, 2-D or 3-D classes of cells. This allows the combinations of cells of different sub-classes often found in Finite Element Analysis (grids of mixed elements) and the data processing is then achieved with a look-up of the appropriate functions of each sub-class.

3.1. Data Fields

A general dataset will consist of points and associated data values in the form of scalar, vector or tensor fields. A **Field** class provides a conceptual definition which encompasses the **PointSet**, **LineSet**, **SurfaceSet** and **VolumeSet** sub-classes. An instance of class Field is a set of data with a given number of points, elementary cells, and field variables. Figure 1 shows part of our class hierarchy, with some of the basic entities we have defined:

The following functions are provided with the definition of **Field**. The sub-classes **LineSet**, **SurfaceSet** and **VolumeSet** take advantage of method inheritance and can re-use all of these:

- Evaluate the type of the cells and their connectivity.
- Create a Postscript description file for hardcopy presentation.

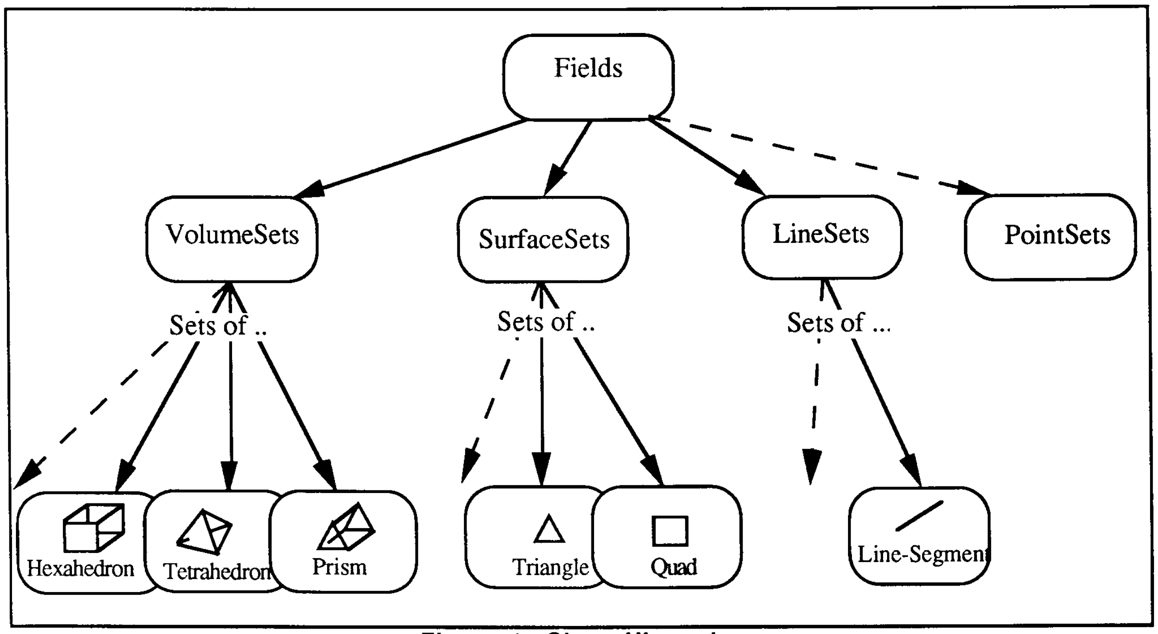

Figure 1: Class Hierarchy

- Display the set as an opaque volume, a wire frame, a cloud of pseudo-colored points.
- Display the set as deformed geometry.
- Clip against geometric objects and display as above.
- Clip against another instance of **Field**.
- Copy, scale, reshape and orient a data glyph at selected points (See [6]).
- Derive gradient, curl, divergence or Laplacian of a field variable.
- Clip the set based on data values or based on spatial coordinates.
- Use the set of points as Point Locators (to initiate particle tracing for example).
- Use the points' coordinates and/or their associated data values.

The core of a data visualization process consists of mappings of the dependent variables to graphical primitives. These mappings are implemented generically by the processing functions of each sub-class of **Field**. When all visualization tools are designed to use and output data of type **Field** (or its sub-types), functional composition, whose goal is to combine visualization primitives into a composite visualization technique, can be readily implemented (see section 4). Instances of the sub-classes of **Field** can either use the functions defined above, or use specialized operations. For example, the **PointSet** class has functions to store, access or modify the coordinates and data values of points and to draw the points in several ways. The other sub-classes have more specialized functions.

3.2. LineSets

A **LineSet** is defined as a set of variables of type **LineCell** with associated field values representing a general 3-D space curve or line. Encapsulated with the definition of this data structure are a few specialized operations, available to all instances of the class, such as:

- Display the polyline as a simple curve, a tube, streamtube, or as a ribbon.
- Compute its length.

3.3. SurfaceSets

The class **SurfaceSet** represents the data structure of a grid made of elementary surface elements of type **SurfaceCell**. A **SurfaceSet** is defined as a set of points with associated field values (a domain in 3-D space), and an inter-element connectivity function (either implicit as for regular grids, or explicit as for FEM meshes) assembling points in the lattice. The set can be formed of multiple disjoint surface patches. For example, a 2-D Finite Difference grid is an instance of the class **SurfaceSet**. We now list a few common operations for such sets:

- Compute iso-contour lines for a selected scalar field (**LineSet** objects).

- Compute streamline profiles based on a velocity vector (**LineSet** objects).
- Decimate or optimize the "mesh" [4, 12, 14].
- Compute the grid's surface area.
- Combine **LineSet** objects into a **SurfaceSet**.
- Apply Texture Mapping techniques on the surface to visualize a data component.

3.4. VolumeSets

The class **VolumeSet** characterizes the data structure of a grid made of elementary volume elements like a tetrahedral mesh for instance. Associated with the volumes are operators such as:

- Compute iso-contour surfaces for a selected scalar field (**SurfaceSet** objects).
- Compute arbitrary cross-sections (**SurfaceSet** objects).
- Compute boundary surfaces (**SurfaceSet** objects).
- Compute streamlines based on a velocity field (**LineSet** objects).
- Compute its volume.
- Volume Render.

At this point, it is easy to recognize that each visualization technique is defined as an operator for each class. These functions can be applied on the data values and are used to create and exchange typed data objects between each other. Functional Composition derives from this careful design of typed data. We focus next on the mappings from **VolumeSets**, to **SurfaceSets**, to **LineSets** which are compositions of techniques. The re-use of all the display and processing functions defined for these data types is at the base of Functional Composition.

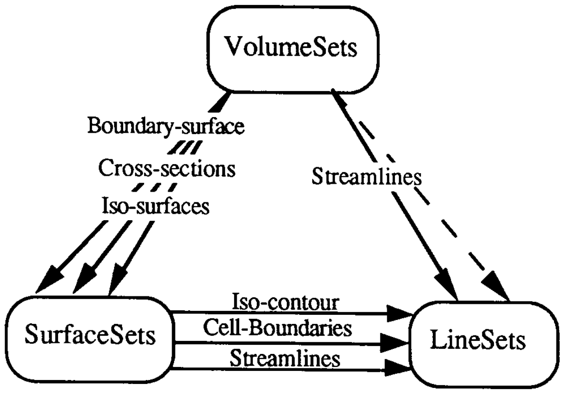

Figure 2: Mappings between Field Objects

4. Composition of Field Operators

4.1. Functional Composition

In our system, all functions are defined as mappings from instances of **Fields** to other instances of **Fields** (or their sub-classes). Figure 2 gives examples of visualization techniques as mappings between variables of the subclasses of **Field** .

Three-dimensional tools such as boundary surface-, cross-section- and isosurface-extraction, which in current systems are limited to creating displayable geometry, are enhanced by creating objects of type **Field** amenable to further enhancing and processing. Each instance of a **Field** encapsulates an underlying geometry and some field data. Thus, functional composition of data visualization tools is made possible by inheriting all the display and processing methods defined for the grid classes. Multiple data extractions and geometry color mappings can be serially composed, each taking a **Field** object and producing another **Field** object.

An example of Functional Composition is to consider a **VolumeSet** V with data fields f_1 through f_n. An intermediate object S of type **SurfaceSet** is created with all of V's data fields stored at its points. Likewise, L of type **LineSet**, inherits the data values of S and all **SurfaceSet** and **LineSet** operations remain available for S and L. For example,

```
S = Cross-Section(V,  F(x, y, z))
/* cross-section surface F(x,y,z) = 0 */
L = Iso-contour-Line(S,  fj,  nb, min, max)
/* isocontour lines fj(X) = min to fj(X) = max */
```

Or more succinctly, to highlight the functional composition taking place:

```
L = Iso-contour-Line(
    Cross-Section(V,  F(x, y, z)),  fj,  nb, min, max)
```

Figure 3 gives the schematic diagram showing the functional composition taking place in our example. **Field** objects are shown in ellipsoids while operators are shown in rectangles. We highlight the fact that each instance of **Field** can be displayed in several ways or passed downstream for further data extraction. Defining and processing the output of all common data visualization tools as **Fields** promotes their functional composition and allow further data processing. This object-oriented approach of designing high level objects for input and output of data visualization tools improves the fan-in and fan-out of processing modules both in the conventional function-based approach and in the new data-flow systems. Data and code reuse are highly favored. The visualization process gains efficiency and practicality by

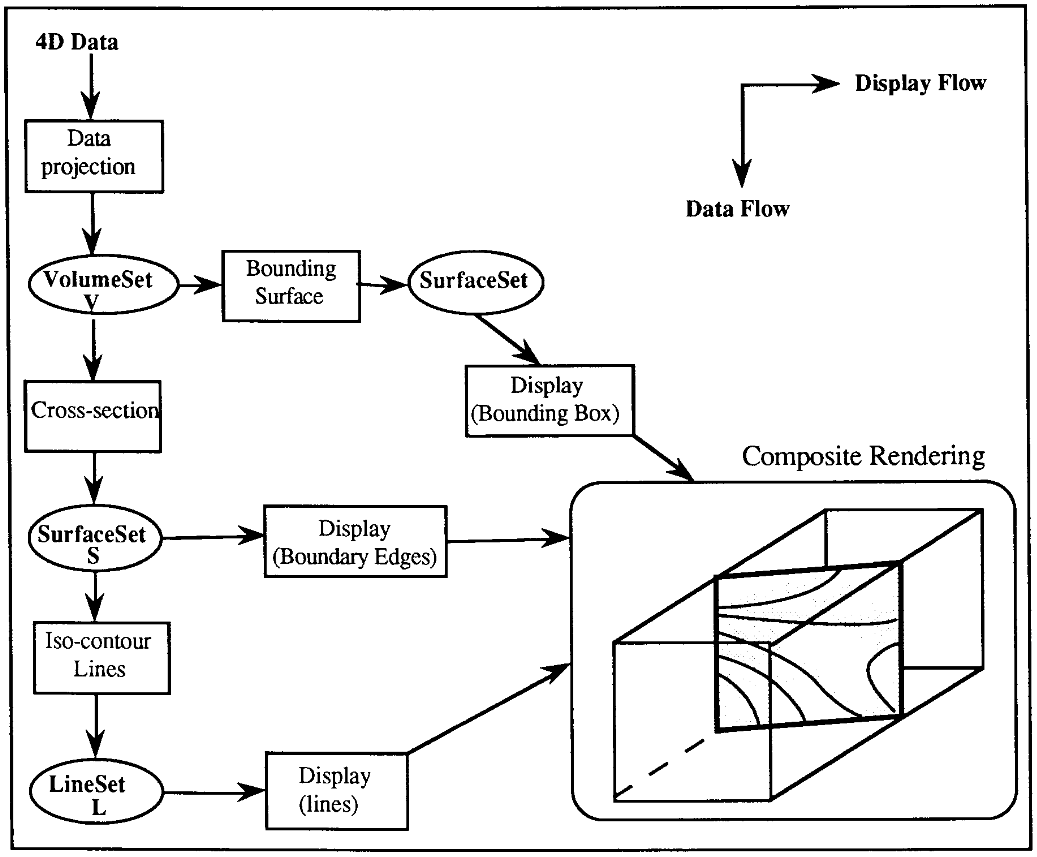

Figure 3: System view of Data Objects and Operations for the given example.

abstracting itself from hardware-oriented graphics primitives. This approach becomes very useful for multi-valued field data sets where we can alternate between different data field views, independently of the underlying geometry and without re-processing of the objects. Computational requirements are thus reduced, promoting interactivity.

4.2. Fields as Domain Selectors

Functional Composition can be interpreted as a data filter of the first operand by the first operator, followed by the application of another technique on the result of the selection. Thus, **Field** objects can provide support for a data extraction operator while remaining partially visible. Transparency may allow the user to see through an object, but the accurate display of multiple objects with various transparency indices is difficult. **Field** objects can be used instead without being displayed. Most often, we will re-use a **SurfaceSet** (such as a cross-section) as a data filter. Drawing its outside line-boundary can help visualize its extent in space, while it is re-used to apply other data extraction techniques, such as contour line drawing, or particle tracing confined to the plane. This functional composition is of great help to the scientist whose goal is to understand inter-relationships between data fields. By providing a way to restrict the domain of application of a data mapping to a sub-set of data of interest, correlation between data fields is more easily extracted and analyzed.

Field objects can also be used as input parameters and their vertices can be used as point locators or seeds for operations like particle tracing, or iso-surface computations.

4.3. Other Operations on Fields

When **Field** instances are regarded as data selectors, it can be appropriate to apply Boolean operations between each instance. Union and intersection are very useful operations that can lead to increased expressiveness.

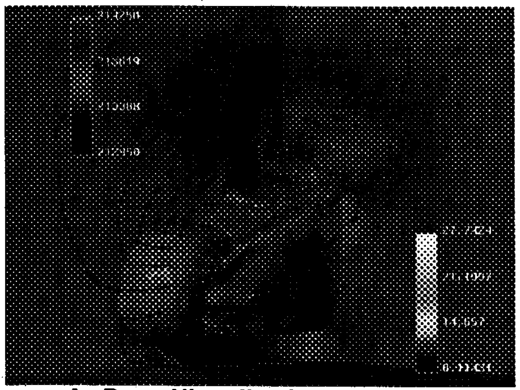

Figure 4: Data Visualization around a high-speed train

These operators can be defined for each sub-class of **Field**, since the underlying data structure allows multiple disjoint sub-sets of elementary cells. It is sometimes necessary to completely remove a part obstructing the view. Cut-aways, which remove features based on the spatial coordinates of their vertices, have also been considered. Since each instance of a **Field** encapsulates an underlying geometry, we can perform Boolean operations on their geometric structures, while conserving all field data. The union of Fields is quite straightforward but is only meaningful if the two objects merged carry the same data information.

Other operations are also available between **Fields**. For example, the tiling of a streamsurface (**SurfaceSet**) with individual streamlines (**LineSets**). Ribbon and streamsurfaces can be constructed thusly. This example shows that we can reverse the natural order of visualization processing shown in Figure 2 which emphasizes going from a higher dimension to a lower dimension. Of particular interest are grid definition techniques which allow us to construct computational grids by scaling, translating and revolving **LineSets** and **SurfaceSets**.

5. Examples

Our library of **Field** classes and operations has been implemented in C++ and runs on SGI workstations. The graphic toolkit Inventor [13] has been used to perform all display operations. Our examples show compositions of data extraction capabilities and highlight the polymorphic display options available to objects of our three most important classes: **VolumeSet** , **SurfaceSet** and **LineSet**.

Our first example in Figure 4 consists of a **VolumeSet** object of 127,049 tetrahedral elements with energy, density and velocity (V_x, V_y, V_z) stored at each

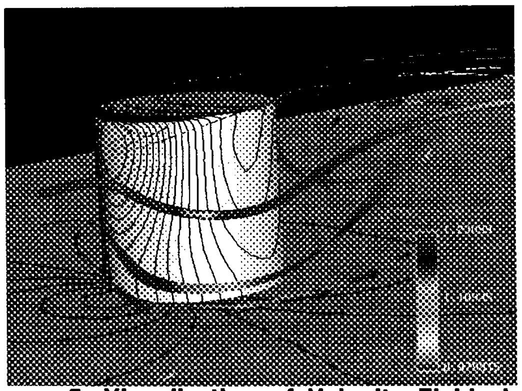

Figure 5: Visualization of Velocity Fields by Polymorphic Rendering

grid point. The object under study is a high-speed train in a flow field. We focus our attention on the front of the vehicle and proceed by calculating an isovalue surface of the x-component of velocity which shows up as a large bulgy surface on the nose of the train. Cross sections along the direction of travel and perpendicularly to the direction of travel are also computed. The addition of the **SurfaceSets** created is then used to show a composition of several data encoding. The cross-sections, the isosurface and the boundary surface are colored with the pressure field (lower left image) and combined to restrict the input domain for the computation of iso-energy lines, shown with a different colormap encoding the variations of the cross-velocity field. Note that the iso-surface actually encodes four scalar variables simultaneously. A pseudo-coloring of the pressure is shown while the iso-lines which are restricted to its surface are colored with a fourth data mapping.

Our second example in Figure 5 shows polymorphic rendering of particle traces computed by a **VolumeSet** object. A cylinder in a transversal flow is studied in a dataset of 246,725 volume cells. The volume object computes its boundary surface on which we compute an iso-pressure **LineSet**. Streamlines are also computed as instances of **LineSets**. As such, they can be displayed in a multitude of ways, without requiring any special-purpose coding. A rake of streamlines is computed and displayed as simple pseudo-colored space curves; another **LineSet** is displayed with velocity vector icons placed at regular intervals, while another one is displayed as a tube and one is displayed as a ribbon, allowing additional color encodings (In this example, the streamline **LineSets** are colored with Pressure). Mappings to pseudo-colored geometric objects can be activated interactively on the various **LineSets** since the data and the geometry are encapsulated in the same data-structure.

The performance of our library of tools provides for interactive inquiries. The train dataset was processed for isosurface and cross-sections in a few seconds, including triangulation and connectivity computations for more than

(See color plates, page CP-35.)

16,000 triangles. The surfaces were then joined, iso-contoured and shaded in a few more seconds on an SGI Crimson. Similarly, the outside surface of the cylinder dataset was extracted and iso-contoured at interactive speed. We have a limited user interface which allows a mix of keyboard inputs and direct manipulation via the Inventor toolkit. Since all objects in the graphics scene contain geometry and data values, computational queries can be readily answered. Each common grid type has "constructor" methods available to read in and store sets of data points and no programming is required. However, the end-user would need to add an additional member function for the given data type for data encoded in a new format.

6. Discussion

Since our visualization algorithms rely on connected components of elementary cells, their implementation requires more work. For example, functions like Marching Cubes iso-surface extraction [7] or iso-contour lines are cell-based by nature and their output is traditionally cast into sets of disjoint geometric entities. Here, to take advantage of the grid data structures and guarantee surface continuity and a consistent right-hand rule ordering of the vertices, surfaces are more easily constructed as a moving front intersecting the volume [15]. Likewise, iso-contour lines must also comply to our design and be fully connected lines, instead of the concatenation of line segments at rendering times. (This is curve sequence contouring versus grid sequence contouring [10]). Note that by construction, streamlines offer naturally connected paths and don't require new implementations. We reap an important benefit from this redesign. Consider iso-level data mappings. There is a well known ambiguity which arises when a cell spans a saddle in the data (Two opposite corners above and two below the threshold value in a 2-D cell). This may lead to spurious holes or surface segments at rendering times, unless additional efforts are invested to carefully handle these cases (See [9] for a very good survey). We have implemented these functions in an advancing front fashion to avoid this ambiguity. An active set of cells is maintained at the boundary of the isosurface and edge and orientation information is passed to the new candidate cells. Because isocontour lines or isosurfaces are constructed incrementally, we obtain local coherency in the numbering scheme of points and elementary cells. When re-used, the **Field** objects are more efficient and they do not require additional connectivity mapping or renumbering.

7. Conclusion

The data extraction operations we show in our examples are not new, but our contribution is to provide the environment where they can be easily combined. We defined elementary cells of various topological dimensions endowed with display and data extraction operators. They are assembled as **Field** objects to represent grids of data. **Field** objects encapsulate data fields with an underlying geometry and offer a rich functionality. They avoid the strong type restrictions typical in other systems by combining display and data manipulation operations. They can be re-used at different stages of the visualization process, thus increasing the fan-in and fan-out of processing modules and enabling better composition of data mappings. We could not achieve similar composite visualization in the other systems we are familiar with, because of their use of pure graphical primitives.

Our data objects can be joined or act as filters to promote the serial composition of visualization techniques. This composition helps understand inter-relationships between data fields and facilitates the analysis of data correlation. Our system design opens new ways to scientific exploration and provides an open-ended architecture for implementing new visual representations whose effectiveness should be evaluated by using principles of visual perception [5].

The structured definitions also allow quantitative analysis to take place. For example, in medical imaging, doctors may want to compute a volume contained between iso-radiation surfaces. In fluid flow, flux computations can be computed through surfaces of interest. Because the objects we manipulate are all piece-wise approximations of well-behaved volumes, surfaces and lines, we can estimate lengths, surface areas, volumes and other numerical values.

We have limited our discussion to grids of linear cells. We intend to expand our library of tools to handle cells with curved boundaries, often used in FEA. We will also focus our effort to data sets in 3D space + time. Constructing streamsurfaces from streamlines or isovalue surfaces from contour lines on successive cross-sections also shows that visualization processes are not limited to mappings to lower dimensions. We plan to research other examples of mappings to higher dimensions.

Acknowledgments

We would like to thank Larry Gritz, Randy Rohrer and Daria Bergen for their valuable help in editing our manuscript, Larry Cannon for his encouragement, and the Department of EE&CS at The George Washington University for its financial support.

References

[1] Butler David M. and Hansen Charles. "Visualization'91 Workshop Report: Scientific Visualization Environments," *Computer Graphics*, 26(3), pp. 213-116.

[2] Foley Thomas A. and Lane David. "Multi-Valued Volumetric Visualization," In *Visualization'91*, pp. 218-225. IEEE Computer Society, October 1991.

[3] Geiben M. and Rumpf M. "Visualization of finite elements and tools for numerical analysis," In *Second Eurographics Workshop in Visualization*, April 1991.

[4] Hoppe Hugues, DeRose Tony, Duchamp Tom, McDonald John, and Stuetzle Werner. "Mesh optimization," *Computer Graphics*, pp. 19-26, 1993.

[5] Ignatius Eve, Senay Hikmet and Favre Jean. "An Intelligent System for Visualization Assistance," To appear in *Journal of Visual Languages and Computing*, 1994.

[6] Kerlic G. David. "Moving iconic objects in scientific visualization," In *Visualization'90*, pp. 124-129. IEEE Computer Society, October 1990.

[7] Lorensen William E. and Cline Harvey E. "Marching cubes: A high resolution 3d surface construction algorithm," *Computer Graphics*, 21(4), pp. 163-169, 1987.

[8] Lucas Bruce et al. "An Architecture for a Scientific Visualization System," In *Visualization'92*, pp. 107-114. IEEE Computer Society, October 1992.

[9] Ning Paul and Bloomenthal Jules. "An Evaluation of Implicit Surface Tilers," *IEEE Computer Graphics and Applications*, 13(6), pp. 33-41, November 1993.

[10] Sabin Malcom. "A survey of contouring methods," *Computer Graphics Forum*, (5), pp. 325-340, 1986.

[11] Schroeder W.J. and Lorensen W.E. "Visage: An object-oriented scientific visualization system," In *Visualization'92*, pp. 219-225. IEEE Computer Society, October 1992.

[12] Schroeder W.J., Zarge J. A. and Lorensen W.E. "Decimation of Triangle Meshes," *Computer Graphics*, 26(2), pp. 65-70, 1992.

[13] Strauss Paul S. and Carey Rikk. "An Object-Oriented 3D Graphics Toolkit," *Computer Graphics*, 26(2), pp. 341-349, 1992.

[14] Turk, G. "Re-Tiling Polygonal Surfaces," *Computer Graphics*, 26(2), pp. 55-64, 1992.

[15] Zahlten Cornelia. "Piecewise Linear Approximation of Isovalued Surfaces," In *Advances in Scientific Visualization*, F.H. Post and A.J.S. Hin (Eds).