# Mix&Match: A Construction Kit for Visualization

Alex Pang and Naim Alper
Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

## Abstract

*We present an environment in which users can interactively create different visualization methods. This modular and extensible environment encapsulates most of the existing visualization algorithms. Users can easily construct new visualization methods by combining simple, fine grain building blocks. These components operate on a local subset of the data and generally either look for target features or produce visual objects. Intermediate compositions may also be used to build more complex visualizations. This environment provides a foundation for building and exploring novel visualization methods.*

**Key Words and Phrases:** interactive, extensible, spray rendering, smart particles, visualization environment.

## 1 Introduction

The diverse needs of scientists demand the development of general purpose, flexible and extensible visualization environments. Flexibility and extensibility are particularly important since no monolithic package can be expected to satisfy every need. Users often need variations on a particular technique and there are always new techniques being developed. In this paper, we present an environment for the flexible creation of visualization techniques from basic building blocks. Designing a technique involves identifying the tasks associated with target feature detection and behavioral responses for displaying those features. This process is simplified by the categorization of these tasks into different classes and the formalization of what constitutes a valid construction. A complete and valid construction defines a new visualization method. Each component of a construction is usually very simple and operates in a local subset of the data space. One of these components specifies which local subset of data to process next. Hence, we can think of these constructions as active processes that can be replicated and sent to work on different parts of the data. In fact, these processes embody particle systems that interact with the data they encounter. Other traditional algorithms can also be decomposed and reconstructed with similar components using this environment.

In the next section, we describe how our work differs from related work. We then describe the Spray Rendering framework and how *Mix&Match* enriches it. This is followed by detailed description of the internals of *Mix&Match*. Finally, we show a couple of constructions and their effects.

## 2 Related Work

In the last few years the data flow paradigm has become popular in scientific visualization. Visualization environments such as AVS [17], Iris Explorer [15], Khoros [12], apE [3], and IBM Data Explorer [10] offer many modules that perform filtering, mapping and rendering tasks that can be combined to achieve a desired visualization goal. These systems offer generality, flexibility, modularity and extensibility. They address the needs of novice, intermediate and expert users. Novices merely load and execute previously constructed networks. Intermediate users use a network editor to construct such a network from existing modules while expert users extend the system by adding modules.

All of these systems can be classified as large grain data flow systems. Data flow refers to the production and consumption of blocks of data as they flow through modules in a network. Modules are required to “fire” as new data arrive. The granularity refers to the size of the data block that the module processes. In these systems, it is the same size as the data model (hence large) rather than being an atomic element of the data model [18]. Granularity may also refer to the size and complexity of the modules. Once again, in these systems they are large in the sense that they implement complete algorithms (e.g. mapping or filter modules).

A drawback with this approach is that memory requirements become prohibitive and cause performance degradation when the data set and the network are large. Performance also suffers when there is a lot of interaction or when the data is dynamic and continually changing. Recently a fine grain data flow environment has been proposed to overcome some of these problems [16]. In this approach, the algorithms are redesigned to work locally on incoming chunks of data where the chunks are a few slices. However, visualization algorithms that require random access to the data set, such as streamlines for flow visualization, are difficult to convert.

In spite of such shortcomings, these systems enjoy a large following mostly because of their flexibility and extensibility to meet new user demands. The importance of these qualities have been recognized in other work. In ConMan, users constructed networks for dynamically building and modifying graphics applications [5]. Abram and Whitted used an interactive network based system for constructing shaders from building blocks [1]. Kass used an interactive data flow programming environment to tackle many computer graphics problems [7]. Corrie and Mackerras recently extended the Renderman shading language to provide a modular and extensible volume rendering system based on programmable data shaders [2].

Our approach strives to maintain the extensibility and enhance the flexibility and interactivity of modular visualization environments at the expense of some efficiency. Instead of modules grinding on entire data sets that flow through them, we send or assign light weight processes to work on a small subset of the data. Thus the two main differentiating points are the granularity of both the modules and size of working data set, and the execution style. Although the visualization of the whole of the data set would computationally be more expensive with this approach, the fine-grained nature of our components which work locally on parts of the data allow quick, interactive exploration. The components are conceptually simple and can be networked in a very flexible way to create more complex components. Because of our choice of execution style, large and dynamic data sets can be handled by localizing these visualization components only to regions of interest.

## 3 Spray Rendering

In this section, we briefly describe the Spray Rendering [11] framework which we use for the construction and application of visualization methods using *Mix&Match*. Spray Rendering uses the metaphor of spray cans filled with smart paint particles. These particles are sprayed or delivered into the data set to highlight features of interest. Features can be displayed in a variety of ways depending on how the paint particles have been defined. To get different visual effects, users simply choose different spray cans from a "shelf". The regions that are displayed depend primarily on the position and the direction of the spray can. Cans also have nozzles that can train the particles into a focused beam or distribute them across a wider swath. The number of paint particles and the distribution of these particles can also be varied.

The key ingredient of Spray Rendering are the smart particles or *sparts*. These sparts are reminiscent of the Particle Systems introduced by Reeves [13] but also possess some of the features of *boids* in [14] and [8]. Sparts are born and have a finite life time. As they travel through the data space, they interact with the data, and perhaps among themselves, leaving behind them visual effects for the users. These particle behaviors can be roughly classified into two categories: *targets* and visual *behaviors*. Targets are features in the data set that the sparts are hunting for (e.g. isovalues, gradients, combination of two fields, etc.) while behaviors specify how sparts manifest themselves visually or non-visually (e.g. leaving a polygon or an invisible marker behind, attaching color attributes, etc.). Some of these effects can be seen in Figure 1.

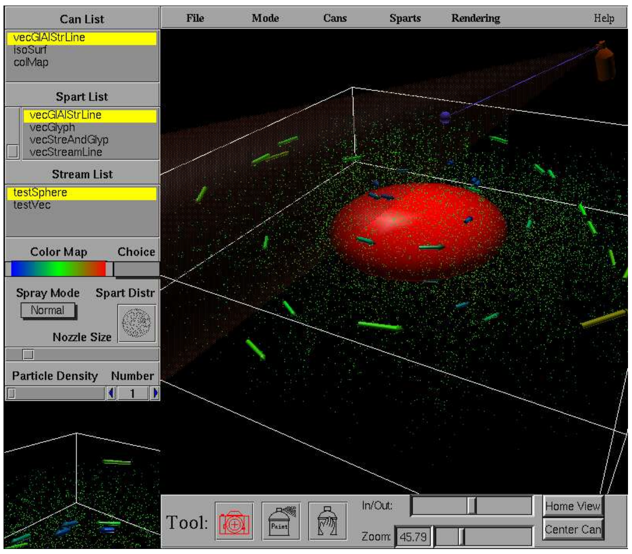


Figure 1: Spray Rendering workspace showing effects of different types of smart particles (*sparts*). Users control viewing and spraying through either graphics window. The lower left graphics window shows the view from the current can.

While sparts are conveniently portrayed to live in 3D space and handle 3D data sets, they can also be designed to operate in lower or higher dimensional space. For example, to track data values from a stationary sensor, one can imagine the spart as sitting on the sensor and producing glyphs (e.g. polylines) according to changes in sensor readings. Or a spart can be called upon to handle time dependent flow fields where the spart is required to travel through time. Eventually, a spart may also map and travel through any N-parameter space. However, we still need to investigate this further since mapping parameter values to Euclidean space will generally produce scattered data sets. This also complicates the point location test for a spart.

In earlier implementations of Spray Rendering [11], we mentioned the idea of mixing different targets and behaviors together to form new sparts. However, we only had predefined sparts in the sense that each spart on the shelf was a complete spart and could not be altered. It was evident that since these sparts shared some common characteristics, they could be decomposed into simpler components and reorganized almost arbitrarily. The next section discusses the issues and implementation details of how this is done.

Note that the idea of visualization processes composed of basic building blocks moving through data does not require spray cans as a launching pad. Indeed, we have a mode where the processes are executed at each grid location.

## 4 Mix&Match

Here we analyze the structure and components of a spart and how they can be categorized. We then discuss the construction rules for building new sparts out of these components. We also address issues such as macro facilities, multi-stage spawning, handling multiple data sets simultaneously and efficiency.

### 4.1 Building blocks of a spart

As can be seen in Figure 1 each spart type produces a different visual effect. Sparts can be programmed to generate iso-surfaces; they can be asked to trace through flow fields and leave vector glyphs, ribbons or stream lines; or generate a quick-and-dirty volume rendering effect by mapping the data values to colored points or spheres. Thus, each spart can be regarded as a different visualization method.

The goal of this research is to provide users the capability to interactively create new visualization methods (or sparts). We achieve this by providing a construction kit, made up of an extensible list of spart building blocks, and allowing users to flexibly combine different pieces together.

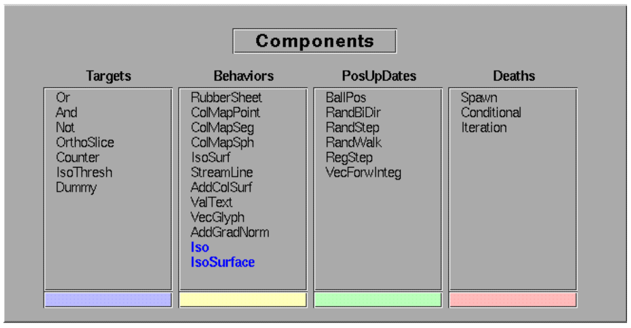
As noted earlier, predefined sparts have two general types of components: target detection and behavioral expression. We can further refine this analysis by noting that sparts are based on particle systems. They therefore have rules regarding when they are born and when they die. In addition, since these sparts are to be sent into the data space, they also have position update rules that may be different from those found in behavioral animation (collision avoidance, group centering, etc.). We have grouped these spart components into four categories:

1. *Targets*. These are feature detection components. They operate on the data locally and check to see whether a boolean condition is satisfied. Components in this category may include local pre-processing operations such as smoothing or gradient operators but not global ones such as Fourier transforms. Relational operators, such as And/Or, are also implemented as target functions and can be used to combine any functions that output a boolean.

2. *Behaviors*. These are components that depend on a boolean condition, usually a target being satisfied, and may produce abstract visualization objects (AVOs) to be rendered.

3. *Position update*. These are components that update the current position of a spart. For example, position changes may depend on the initial spray direction or may be dictated by a flow field.

4. *Birth/Death*. These components decide whether the spart should die or spawn new sparts. For example, a spart may be terminated as soon as a target is found or wait until it has exited the data space.



**Components**

| Targets | Behaviors | PosUpDates | Deaths |
|---|---|---|---|
| Or | RubberSheet | BallPos | Spawn |
| And | ColMapPoint | RandBiDir | Conditional |
| Not | ColMapSeg | RandStep | Iteration |
| OrthoSlice | ColMapSph | RandWalk | |
| Counter | IsoSurf | RegStep | |
| IsoThresh | StreamLine | VecForwInteg | |
| Dummy | AddColSurf | | |
| | ValText | | |
| | VecGlyph | | |
| | AddGradNorm | | |
| | **Iso** | | |
| | **IsoSurface** | | |

Figure 2: Components browser showing the list of functions categorized under targets, behaviors (visuals), position update and death functions.

Figure 2 shows a growing list of components under each category. Each element in the list is a building block that can be used in the creation of a spart. By breaking down the spart into components, we allow the components to be used in the rapid prototyping of other sparts.

Each building block in the construction kit is a regular C function with variable number and type of inputs and outputs. The input and output ports can be connected together interactively. There is strong type checking at the I/O connectivity but no type coercion. Apart from the number and types of inputs and outputs, components may also have parameters that can be set by the user through widgets (e.g. threshold value, step size, etc). A spart is therefore a set of functions grouped together to carry out a specific visualization method.

## 4.2 Putting them together

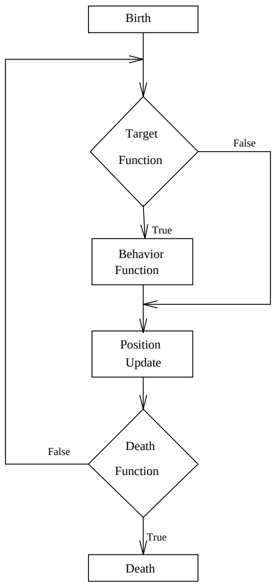


Figure 3: Flow chart illustrating the life-time of a typical spart.

The process of creating the spart corresponding to a visualization method can be seen as the mixing of different pigments on a palette to obtain a desired color. In this analogy, the building blocks are the pigments. We call this process of mixing different building blocks to obtain a desired visual effect *Mix&Match*. The rules for putting the building blocks together are quite simple. The basic pattern follows the typical operations over the lifetime of a spart as illustrated in Figure 3. Note that Figure 3 is merely illustrative. For instance, there may be sparts that do not have a target function and whose behavior function executes unconditionally, or there may be death functions that depend on multiple conditions.

We provide both a textual and a graphical interface to the process of composing a spart and users can switch freely between the two. Spart construction starts with the selection of components to be included in the editor from the components browser (Figure 2). As building blocks are included in the construction, users must manipulate the input and output fields of each component to establish connections. In the textual interface, this is done by editing the inputs and outputs such that a name given to an output field of a component and the input field of another component indicates a connection between them. In the graphical editor (Figure 4), one merely selects the input and output names from the menus of the components in question. We have avoided the use of a programming language for the definition of a spart to make the task simpler for the scientist.

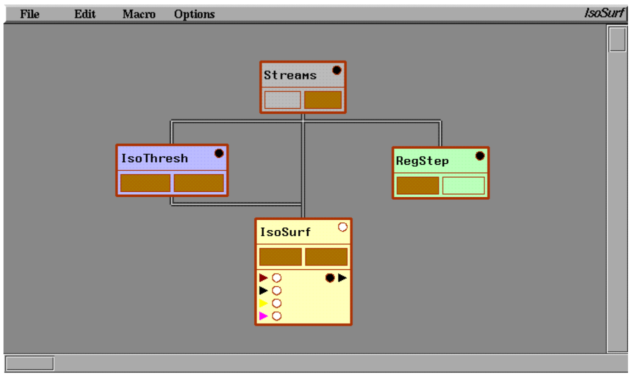


Figure 4: The graphical spart editor showing the iso-surface spart composition. The IsoSurf component is shown expanded to reveal the types and the status of connections. Circles indicate connections of the fields while colors of the triangles represent the types of the fields.

Before enumerating the rules for constructing a spart, let us look at how a common visualization method can be expressed in the form of a spart as it would appear in the textual spart editor. The Marching Cubes algorithm [9] can be converted to a localized spart construction using three building blocks as illustrated below. Instead of looking at every cell in the volume, individual sparts handle a local subset of the data. In this particular example, it would be those cells that the spart visited as it travels through the data space.

*Iso-surface spart:*
```
  IsoThresh [ S1 ] ( Found ) ( Tag ) ( IsoVal )
  IsoSurf [ S1 ] [ Found ] [ Tag ] [ IsoVal ] ( Obj )
  RegStep [ S1 ]
```

The above construction consists of a target function **IsoThresh**, a behavior function **IsoSurf**, and a position update function **RegStep**. The **Streams** component (Figure 4) exists in all networks for binding data streams to the input ports of the other components. There is also a default death function that kills the sparts once they exit the bounding box of the data set. Input fields are identified with [ ] while output fields are identified with ( ). **IsoThresh** is a simple function that examines the cell the spart is in within the input stream **S1**. It sets the boolean **Found** if there exists a surface at the given iso-value. In this case, **IsoSurf** will generate one or more polygon visualization objects **Obj** in that cell. The spart then advances a fixed step size according to the parameter set in **RegStep**. These functions are repeated until the spart is terminated after exiting the bounding box of the data volume.

The power of *Mix&Match* becomes evident when the user has the flexibility of modifying sparts to produce different effects. For example, in the construction above, the death function could be made conditional on an iso-surface being found. This slight modification will produce iso-surfaces that are visible only from the spray can's perspective. The position update function may be modified to follow surface gradients. Likewise, the behavior function may be substituted with one that paints the entire cell achieving a *cuberille* effect[6].

For some sparts it may be undesirable to rely on position update and death functions that sample the data. In the example above, cells that are missed will not produce polygons and may result in discontinuous surfaces. For this reason, we provide a mode where the spart visits all the cells and only the target and behavior functions are executed.

There are a few simple rules for constructing sparts which are enforced either during construction or during parsing:

1. *Strong typing.* The types of the input and output fields of a connection must match. Type checking is done at the time the connection is being established in the graphical editor and is postponed until parsing time in the textual editor.
2. *No optional inputs.* All function input fields must either be connected to an output field or have a constant value associated with it. Output fields can be left floating.
3. *Fan out but no fan in.* There can only be a single connection into an input field. The same output field can be connected to multiple input fields however.
4. *Acyclic graph.* A directed graph where the edges denote the dependency between components has to be acyclic.
5. *Execution order.* The components of a spart execute according to a specific order: target functions first, then behavior functions, position update functions and finally birth/death functions. Topological sorting ensures correct dependency ordering within each category. However, a component from a category that will execute earlier should not depend on another component that will execute later.

The environment also provides a macro facility to build a component from a collection of other components allowing more succinct compositions. The macros can be nested and more than one macro can appear in a composition. Note that macros are like procedures and can be saved and used in compositions. They are not merely a temporary visual grouping of components.

## 4.3 Handling multi-parameter data sets

A spart can handle multi-parameter data sets. Each parameter of a multiple parameter data set is treated as a separate data stream. The spart composition then contains separate components to handle different data streams individually. The stream identifiers, e.g. **S1**, saved with the spart composition are bound by the user to the actual data streams at the time the spart is loaded into a can. When two different streams appear as input in the composition, it may mean that they are the two parameters of a data set with the same bounding volume, or two different data sets with different bounding volumes. The latter implies that there may be multiple incarnations of the spart, one in each stream. An incarnation in one stream may be dead but another may still be alive and the spart will continue executing its program until all incarnations are dead. This allows us to look for relationships between parameters of the same data set or between parameters in different data sets that have overlapping bounding volumes.

Relational expressions used in combining different targets, whether from the same data stream or not, are also implemented as target functions. For example, if the target functions **TargetA** and **TargetB** have boolean outputs **A** and **B**, they could be combined as follows:

```
TargetA ... ( A ) ...
TargetB ... ( B ) ...
And [ A ] [ B ] ( AandB )
```

### 4.4 Multi-stage spawning

Sparts are initially spawned as they are sprayed from the cans. Each time the user sprays or holds the spray button down, sparts are continually being spawned and added to the can's pool of sparts to be executed. These sparts are eventually executed and terminate when they have satisfied the death function.

New sparts may also be spawned during the life span of a spart. This is achieved by including a spawn function in the construction. The spawn function takes the name of the spart to be spawned as an argument. The new spart does not have to be the same as the parent spart. The spawn function is handy in certain situations. For instance, new sparts may be spawned in the vicinity where iso-surface sparts have located a surface. This will fill in the surface more quickly than relying on the spraying marksmanship of the user.

### 4.5 Extensibility

One advantage of *Mix&Match* over other systems is the relative ease of writing small fine-grained functions that perform very specific tasks (e.g. update position in certain way or produce certain AVOs). In comparison, coarse-grained modules are typically larger and also have some degree of code replication since some modules may be very similar in certain tasks but differ on details.

Extending the functionality of the environment involves adding more functions to the browser. New functions must be registered so that they can be included in the browser. A configuration manager provides a graphical user interface for this task. The user defines the number and types of the inputs and outputs and graphically designs the control widgets for the component. The configuration manager then generates appropriate wrapper code. The new component is integrated into the system by compilation and linking.

### 4.6 Efficiency and object compaction

There is a tradeoff between flexibility and efficiency. If components exist at a low level, there is greater flexibility in composition but one suffers higher costs in execution overhead. Inversely, a high level component results in loss of flexibility. At the cost of code replication and program size, one can include both the high level module and its components. At the extreme, one could have the spart be a single module. We call these predefined sparts. If a certain spart is to be used often it may be worth the effort to re-implement it as a predefined spart.

The building blocks are written independently from each other and hence have to determine at run time where to find the inputs and the parameters they need and where to send the outputs. This is handled by the components looking for their inputs and parameters from fixed places in their own structure. During parsing, memory is allocated for the addresses of input and output fields of each component. These addresses are filled according to the connections in the composition. All that the function does when called is to dereference the pointers from the component structure passed to it. Multiple instances can thus coexist in a composition.

The main reason for the cost in execution is not so much the extra function calls and pointer dereferences but the fact that those functions that produce AVOs have to generate them at each call that satisfies the target function. For instance, a predefined streamline spart would accumulate vertices that define a single multi-segmented line object (polyline). A *Mix&Match* streamline spart, on the other hand, would define a simple line segment consisting of the present and the previous vertex each time it is called. This causes inefficiency both in execution (many more calls to malloc) and in storage (inner vertices are replicated). The rendering time also suffers because of the greater number of AVOs generated that need to be traversed. To alleviate the latter problem, objects of similar attributes are compacted periodically into a single object. In the above example, all the simple line segment objects would be compacted into a single polyline object.

## 5 Examples

In this section, we give some examples of spart compositions. By changing single lines of these compositions, different visualizations can be achieved. Users can experiment with the different compositions and save those that they are likely to use again.

### 5.1 Flow visualization

Showing streamlines is a typical flow visualization method for displaying vector fields (Figure 5). In this technique, the path of a massless particle through the flow field is traced assuming that the vector at the current location is tangential to the path. The new position is calculated by forward integration using the vector at the current location. Such a spart can be constructed as follows:

*Streamline spart:*

```
  StreamLine [ S1] [ =TRUE ] ( Vec ) ( Obj )
  VecForwInteg [ S1 ] [ Vec ]
```

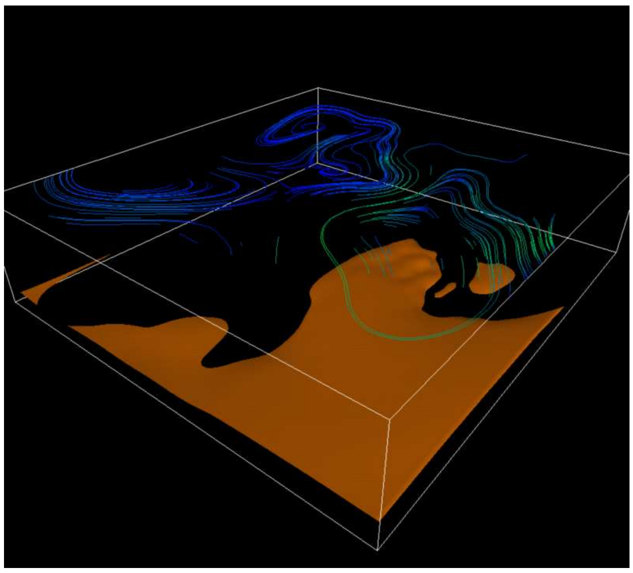


Figure 5: A spart that generates streamlines from a vector field. Iso-surfaces from a scalar field are also shown in the background.

This is a spart without a target function. The first component is a behavior function that unconditionally outputs objects while the position update function **VecForwInteg** calculates the new position. The first component also outputs the calculated vector at the current location so that it can be used by the following component. It is a good idea to pass intermediate values that may require expensive computation so that other components can use them without recomputation.

Another technique for vector field visualization is to use vector glyphs. Usually, the glyphs are placed at some sub-sampling of the grid but in spray rendering, we can place them at intervals along the path of the spart. By replacing the behavior function in the composition above with a behavior function that produces vector glyphs, we can place glyphs at intervals along a streamline. Alternatively, we can include both behavior functions and obtain streamlines and glyphs along the streamline.

## 5.2 Iso-surfaces

We can make some minor variations to the iso-surface spart described in section 4.2. For example, we can combine two or more iso-surface seeking sparts within one construction. The target functions may be bound to the same input stream (i.e. looking for different iso-values) or they may be bound to different input streams. The target function of the iso-surface spart can be used in a spart that does not actually generate an iso-surface, but merely uses this component as a filtering operation. Another behavior that takes in a geometry(the iso-surface) as input and colors it according to a stream value can be used to investigate a relationship between two parameters of a data set by showing the variation of one parameter over a surface on which the other parameter is constant as in [4]. The following composition illustrates these ideas:

*A spart with four streams:*
```
IsoThresh [ S1 ] ( Fnd1 ) ( Tag1 ) ( Val1 )
IsoSurf [ S1 ] [ Fnd1 ] [ Tag1 ] [ Val1 ] ( Obj1 )
AddColSurf [ S2 ]  [ Fnd1 ]  [ Obj1 ] ( Obj2 )
IsoThresh [ S3 ] ( Fnd2 ) ( Tag2 ) ( Val2 )
VecGlyph [ S4 ] [ Fnd2 ] ( Vec )
RegStep [ S1 ]
```

In this example, an iso-surface is created based on one stream (S1:geopotential height) and the values of another stream (S2:humidity) are mapped onto the generated surface as color. A third stream (S3:temperature) is filtered based on a threshold value (S4:wind field) and vector glyphs are placed at those locations that satisfy this condition.

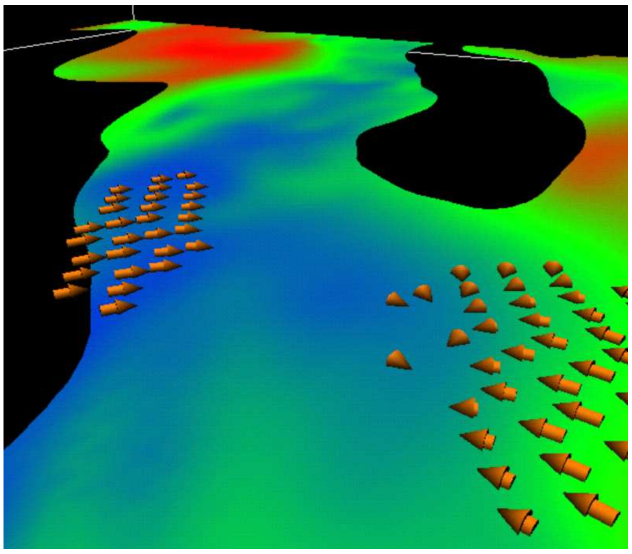


Figure 6: A spart that shows the relationship between four input streams of a climate model. An iso-surface is generated from the geopotential height field and the relative humidity is mapped onto this surface. The temperature field is thresholded and wind vectors placed at the locations that would have produced an iso-surface.

## 6 Conclusions

*Mix&Match* is an extension to Spray Rendering which allows composition of visualization techniques from simple, fine grain building blocks. Unlike the predefined sparts presented in our earlier work, the *Mix&Match* sparts are made up of elementary components and users are allowed to edit them by adding, removing or changing different components with the aid of a textual or graphical spart editor. This capability encourages the users to experiment with different ways of visualizing their data. In contrast to data flow networks, the execution model used here sends multiple independent agents to different localities of the data space. Its strengths are its extensibility and the fact that users can create their own visualization methods interactively. On the other hand, its weaknesses are primarily efficiency and the duplication of effort by multiple sparts that enter the same data space.

The current work opens up the proverbial Pandora's box. There are many issues that need to be resolved to fully exploit the capabilities of sparts. Among these are the traversal through unstructured grids and scattered data, mapping to parallel architectures, inter-spart communication and letting sparts query scientific databases. Whether our approach offers advantages in massively parallel environments is something that we will be investigating in the near term.

## Acknowledgements

We would like to thank the other members of the spray team: Jeff Furman, Tom Goodman, Elijah Saxon, and Craig Wittenbrink. We would also like to thank Dr. Teddy Holt and Dr. Paul Hirschberg for kindly providing us the meteorological data sets used in the figures. Support for this work is partly funded by NSF grant CDA-9115268 and ONR grant N00014-92-J-1807.

## References

[1] Gregory D. Abram and Turner Whitted. Building block shaders. *Computer Graphics (ACM SIGGRAPH Proceedings)*, 24(4):283 – 288, August 1990.

[2] Brian Corrie and Paul Mackerras. Data shaders. In *Proceedings: Visualization '93*, pages 275 – 282. IEEE Computer Society, 1993.

[3] D. S. Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60 – 69, 1990.

[4] T. A. Foley and D. A. Lane. Multi-valued volumetric visualization. In *Proceedings: Visualization '91*, pages 218 – 225. IEEE Computer Society, 1991.

[5] Paul E. Haeberli. ConMan: A visual programming language for interactive graphics. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):103 – 111, 1988.

[6] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computer tomograms. *Computer Graphics and Image Processing*, 9(1):1 – 21, 1979.

[7] Michael Kass. CONDOR: Constraint-based dataflow. *Computer Graphics (ACM SIGGRAPH Proceedings)*, 26(2):321–330, July 1992.

[8] G. David Kerlick. Moving iconic objects in scientific visualization. In *Proceedings: Visualization '90*, pages 124 – 130. IEEE Computer Society, 1990.

[9] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163 – 169, 1987.

[10] B. Lucas, G. Abram, N. Collins, D. Epstein, D. Gresh, and K. McAuliffe. An architecture for a scientific visualization system. In *Proceedings: Visualization '92*, pages 107 – 114. IEEE Computer Society, 1992.

[11] Alex Pang and Kyle Smith. Spray rendering: Visualization using smart particles. In *Proceedings: Visualization '93*, pages 283 – 290. IEEE Computer Society, 1993.

[12] J. Rasure, D. Argiro, T. Sauer, and C. Williams. Visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology*, 2(3):183 – 199, 1990.

[13] W. T. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. *Computer Graphics*, 17(3):359 – 376, 1983.

[14] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25 – 34, 1987.

[15] G. Sloane. *IRIS Explorer Module Writer's Guide*. Silicon Graphics, Inc, Mountain View, 1992. Document Number 007-1369-010.

[16] Deyang Song and Eric Golin. Fine-grain visualization algorithms in dataflow environments. In *Proceedings: Visualization '93*, pages 126 – 133. IEEE Computer Society, 1993.

[17] C. Upson. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30 – 42, 1989.

[18] C. Williams, J. Rasure, and C. Hansen. The state of the art of visual languages for visualization. In *Proceedings: Visualization '92*, pages 202 – 209. IEEE Computer Society, 1992.