University of California
Santa Cruz

Mix&Match : A Construction Kit for Scientific Visualization

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

by

Naim Alper

March 1995

The dissertation of Naim Alper is approved:

Dr. Alex Pang

Dr. Patrick Mantey

Dr. Glen Langdon

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Mix&Match : **A Construction Kit for Scientific Visualization**

Naim Alper

ABSTRACT

A new, modular and extensible scientific visualization environment is presented. It provides the ability for a user to define a visualization technique from basic components through visual programming. Unlike data-flow based modular visualization environments, where the network defines a program and the end result is the visualization, the composition in *Mix&Match* defines the overall behavior of a tool. Different tools can be used to visualize the same data in different ways. High level mode-settings allow different modes of interaction without necessarily requiring changes in composition. This greatly enhances the level of interactivity and emphasizes the exploratory nature of scientific visualization. Another distinguishing feature of the environment is the finer granularity of the components. Finer granularity allows greater flexibility in composition and results in a rich collection of techniques. The components are simple and small, permitting new ones to be readily added to the system, thereby encouraging the exploration of new visualization techniques. This is facilitated by an easy to use configuration manager. The tool behaviors are based on the same simple, particle based, template which acts as a unifying representation for visualization techniques. In addition to the development of the environment, traditional techniques are decomposed to fit this template and new visualization techniques are developed.

Keywords: scientific visualization, spray rendering, toolkit, visual programming

Preface

Acknowledgements

Foremost, I would like to thank my advisor Prof. Alex Pang for maintaining his faith in me and his constant encouragement and guidance. His idea of Spray Rendering provided fertile grounds for this work to flourish. He provided a working environment that was free of stress and pressure which I appreciated greatly.

I would like to thank Prof. Glen Langdon and Prof. Pat Mantey for agreeing to be on my reading committee despite their heavy work loads and for their helpful comments on the dissertation draft. Special thanks are due to Prof. Pat Mantey for securing the ONR grant for the REINAS project that provided my funding for this research.

My thanks to other members of the REINAS visualization team, Tom Goodman, Craig Wittenbrink, Jeffrey Furman and Elijah Saxon, for their contributions and team spirit. Many thanks to Dr. Wendell Nuss and Dr. Paul Hirschberg of the Naval Postgraduate School for providing the climate model data. Thanks also to fellow graduate students Cathi Colin, Shankar Ramamoorthy, Madhukar Thakur and Judy Challinger for sharing their thoughts on the joys, trials, and tribulations of graduate student life.

I would like to thank Kadriye Ercikan for being supportive throughout and for her love and inspiration. Having already gone through this process, she was always full of great advice. Thanks also to my daughter, Deniz, for providing the opportunity to occupy myself with her delightful development and to leave work at the workplace.

Finally, I would like to thank my parents and my sisters in Cyprus who have had to endure long years of separation so I can continue my education. I dedicate this work to them.

Publication History

Some early design ideas were discussed in

A. Pang, N. Alper, J. Furman and J. Wang.
Design Issues of Spray Rendering.
Proceedings: Compugraphics '93, pages 58 – 67.

The gist of the dissertation was presented in

A. Pang and N. Alper.
Mix&Match: A Construction Kit for Scientific Visualization.
Proceedings: Visualization '94, pages 302 – 309.
IEEE Computer Society, 1994.

The techniques used in the *BumpMap* component were presented in

A. Pang and N. Alper.
Bump Mapped Vector Fields.
IS&T/SPIE Symposium on Electronic Imaging: Visual Data Exploration and Analysis, volume 2410,
February 1995.

1. Introduction

Scientists are producing ever increasing amounts of data through measurement or simulation. Since it is difficult to extract information from raw numeric data, visualization has become the normal procedure for the analysis and interpretation of data. Visualization takes advantage of the wide bandwidth of the human visual system: a simple graph or a color coded graphical image provides more immediate information than can be gained by perusing numbers. The field of computer science that has grown with the advent of computer graphics techniques and the wider availability of color workstations has come to be known as *Scientific Visualization*. It has grown in importance in recent years, particularly after the impetus provided by the NSF panel report which identified areas in need of research and funding[McC87].

Up until the late 1980's, there existed no general software package to meet the diverse needs of scientists from many disciplines. Scientists used many different programs to achieve specific visualization goals. Some domain-specific modular and extensible software packages evolved that included all the usual techniques encountered in a particular discipline[BMP+90]. However, in the last few years a data flow paradigm has become very popular for general purpose, modular and extensible visualization software. Products such as AVS[Ups89], Iris Explorer[Slo92], apE[Dye90], and IBM Data Explorer[LAC+92], which have also come to be known as Modular Visualization Environments (MVE), all use this approach.

The data flow paradigm offers a good mapping to the problem domain: acquired data is perhaps filtered, then mapped to some visual parameters and the result rendered. These packages provide many modules that perform such filtering, mapping and rendering tasks that can be combined to achieve a desired visualization goal. However, it is their modular and extensible design and relative ease of use that has made them attractive. Flexibility and extensibility are particularly important to meet the diverse needs of scientists since no monolithic package can be expected to satisfy every need. On the other hand, flexibility and extensibility in these environments are limited by the coarse granularity of the modules, and very little direct user interaction is supported.

Recently, Pang and Smith have proposed a new paradigm for scientific visualization[PS93a, PS93b]. The name *Spray Rendering* reflects the metaphor used to describe the way the scientist interacts with the data. The user has "spray cans" available which can be filled with different "paints". As the user directs and delivers a dose of "spray", the paint interacts with the data and makes them visible. By choosing different paints from a palette and orienting the spray can to specific areas, the scientist is able to explore the data set visually. The paint particles in this paradigm are "smart" in their behaviors. That is, they can search for target features in the data set, manifest themselves visually in various ways, and interact with each other as well as the data.

The motivation for this dissertation is to develop a software system and framework for user interaction that overcomes the three shortcomings of the MVEs. A scientific visualization environment called *Mix&Match* is developed that maintains the modularity and graphical interface of MVEs but enhances their interactivity, flexibility and extensibility by allowing visualization tools to be composed from basic building blocks through visual programming. The behaviors of these tools are modeled on spray rendering.

Interactivity, flexibility and extensibility are important and desirable since they help scientists in their goal of gaining insight into their data sets. Visualization is an exploratory process and direct interactivity emphasizes and facilitates that aspect. There is a direct cause and effect between the actions of the scientist and the visualization. Greater flexibility in connecting components permits scientists to experiment with new compositions, and easier extensibility encourages experimentation with new techniques.

Direct interaction is achieved by using the tool (the spray can) to interact with the data, and benefits novice users who merely use existing tools. Users can manipulate the spray can within the data set and see the effects immediately. High level mode-settings allow the same composition to result in different interactions. In one mode, for example, the spray can becomes a probe that can be used to interactively explore the data set. In another mode, the spray can plays a passive role and is not instrumental in the visualization. Hence, although it is modeled on spray rendering, the idea of components of fine granularity operating locally is independent of spray rendering.

Other modes allow simple animations. These modes, in effect, behave as control components but do not appear in the composition where they would increase the complexity and would be harder to use. Complex visualizations can be obtained by using multiple tools multiple times. Yet, the networks defining each technique are simple and separate from each other. This is in contrast to the data-flow environments where, as the complexity of the visualization increases, the complexity of the network that generates it increases.

The components are fine-grained in that they are simple and act on a local neighborhood of the data set. The fine granularity permits greater flexibility in composition, thereby producing a richer set of visualization techniques. Instead of a visualization technique being encapsulated in a single module, as happens in coarse-grained MVEs, the technique can be decomposed into smaller components which can be used in other compositions. Greater flexibility benefits users who experiment with tool construction.

The simplicity of the components allows new components to be readily added to the system so that extensibility is enhanced as well. Many visualization techniques differ in minor details and the addition of a simple component enriches the repertoire of the system. Users are thus encouraged to experiment with new visualization techniques. Component writers are the immediate beneficiaries of easier extensibility.

As well as providing the software environment, many of the popular visualization techniques have been decomposed into components and adapted to work in *Mix&Match* as part of this thesis research. The techniques are made to fit the same particle based template and thus the environment provides a unifying representation for visualization techniques. For an end user, all the tools essentially behave in the same way, which makes them easier to understand. The template facilitates tool composition which consists of the filling in of the template. It also makes it more likely that a component needed for a particular technique will already exist.

Finally, *Mix&Match* has been used successfully to investigate and experiment with new visualization techniques. As an example, the computer graphics technique of bump mapping was used in various ways to visualize vector fields[PA95].

In summary, the contributions of this dissertation are:

- the design and implementation of a new scientific visualization environment, based on fine-grained components, that is modular and extensible which offers direct interactivity, greater flexibility and easier extensibility
- the provision of a unifying representation for many visualization techniques
- the development of new visualization techniques using the environment

The rest of the dissertation is organized as follows. In chapter 2, the different categories and examples of scientific visualization environments are discussed. An overview of scientific visualization techniques is given in chapter 3. In chapter 4, the design and architecture of *Mix&Match* is presented, and implementation details are covered in chapter 5. Chapter 6 lists the components that have been implemented and provides example compositions and visualizations. Finally, conclusions and future work are presented in chapter 7.

2. Scientific Visualization: Environments

Mix&Match is a scientific visualization environment and needs to be discussed in the context of other environments. In this chapter, a brief survey of current examples of scientific visualization environments is presented with more emphasis on the ones that are related to *Mix&Match*. *Spray Rendering*, as originally envisaged by Pang and Smith[PS93a, PS93b] is also presented, since it was used to model the components. How *Mix&Match* relates to other visualization environments and spray rendering is then discussed at the end of this chapter.

2.1 Some Scientific Visualization Environments

Many commercial and public domain packages have been developed since they were first proposed by the NSF Panel on Graphics, Image Processing and Workstations[McC87]. The panel's report pointed to the need for general purpose scientific visualization environments as opposed to domain specific, in-house developed, and monolithic systems.

There are basically three categories of visualization software:

1. **Graphics Libraries.** At the lowest level, one can regard graphics libraries, such as Iris GL, Dore, and PHIGS, as visualization software since they provide the means for writing programs to view and analyze data. Although the use of graphics libraries offers the greatest flexibility, it suffers from the disadvantage that a great amount of time needs to be invested in writing and supporting code. From a scientist's perspective, this is uneconomical and unattractive.

2. **Turnkey Visualization Systems.** In this category, most of the work has been done for the end user such that no programming is required to obtain results. However, as the name suggests, the systems cannot be modified and extended and they do not satisfy every need.

3. **Extensible Visualization Systems.** These systems can be used to extend and customize an application. They are application builders that recognize the fact that

there will always be needs that are not foreseen. This category can further be refined into two subcategories:

- *Programming Library Systems.* These systems provide a programming library and a high level command language to construct customized applications.
- *Modular Visualization Environments.* These are modular environments that offer a data-flow visual programming interface for achieving a visualization goal. This category has become very popular because it is flexible and caters to users of different levels of expertise.

2.1.1 Turnkey Visualization Systems

- **The Data Visualizer,** *Wavefront Technologies Inc.* The Data Visualizer offers a traditional, menu-driven application interface and primarily deals with 3D data[BAWW90, Mer91, Bel93]. Version 2.1 also supports a command language interface. The architecture is tool-oriented where each tool is an instance of a visualization method. A *Visualization Tool Manager* accepts input from the user and manages the creation and manipulation of various tools. Some of the tools provided are probes, cutting planes, isosurface tools, particle emission and point volume tools. Interaction consists of selecting tools from a menu which can be turned on and off. The Data Visualizer handles various kinds of grids. Its native file format is an ASCII format called *wave* and it supports custom file format readers written by users. The user interface is also customizable. Keyframe and flip-book animations are possible.
- **Fieldview,** *Intelligent Light.* Fieldview is primarily for fluid dynamics data[Leg91]. Its native data format is the same as that of PLOT3D, five dimensionless quantities on a 3D grid. Calculator tools are provided that can compute compound functions (scalar and vector) on the data and display the results as cutting-planes, iso-surfaces, streamlines, particles and other techniques. It has scripting and animation for video production and can run on low-cost platforms using the IVIEW-DORE graphics

library. The most recent version provides a direct interface to commercial CFD solvers and has an open programming interface.

- **Spyglass Dicer,** *Spyglass Inc.* This is a 3D manipulation program that permits the examination of large data sets through the viewing of slices and sections. Images can be stacked together to simulate volume rendering. Isosurfaces can be combined with cutting planes and the motion of cutting planes and blocks can be animated.
- **VoxelView,** *Vital Images.* VoxelView is an interactive volume rendering software system that has a high speed opacity blending algorithm implemented in microcode on Silicon Graphics machines. Only rectilinear grids are supported. The Ultra product can mix geometry with volume data.
- **VolVis,** *SUNY Stony Brook.* This is a framework for volume visualization algorithms[ASK92, AHH$^+$94]. It is supported by a generalized abstract model which provides for both geometric and volumetric constructs. The techniques supported include a fast volume rendering algorithm as well as costly, realistic ray-tracing. Tools include 3D manipulation, key-frame animation and quantitative analysis.
- **Vis-5D,** *University of Wisconsin.* This is a public domain package for the visualization of 5-dimensional data sets[HS90]. The dimensions refer to three spatial dimensions, one time dimension and a dimension for multiple parameters to be visualized. It is popular among meteorological scientists. It offers animated isosurfaces, 2D slices and streamline traces.

Some other turnkey systems include **SciAn** from Florida State University that is mainly for environmental visualization, **BOB** from the Army High Performance Computing Research Center and **VoxelBox** from Jaguar Software for MS Windows.

2.1.2 Extensible Visualization Systems

Programming Library Systems

- **PV-Wave,** *Visual Numerics.* This is a two and three dimensional visualization package that uses a command language interface (although the newer Point & Click version has a menu-driven interactive graphical interface[Kri91]). PV-Wave provides an intelligent *data previewer* for importing ASCII data files. It offers a macro tool for automating repetitive tasks. True ray-cast volumetric rendering that can be combined with geometric objects is provided as well as iso-surfaces. PV-Wave Advantage extends the programming library with the IMSL libraries, making it the most extensive scientific programming interface available. Another version connects to a user interface builder.

- **IDL,** *Research Systems Inc.* IDL or Interactive Data Language is an array-oriented language for the analysis and visualization of scientific data. It can be used interactively or it can be used to create functions, procedures and applications.
- **FAST,** *NASA Ames.* FAST is a public domain program developed at NASA Ames for the visualization of fluid dynamics data[BMP+90]. It is a collection of programs communicating through Unix sockets. A central hub process manages a pool of shared memory. A collection of libraries and utilities are provided for building application modules. FAST offers the usual isosurface and stream line modules as well as a calculator module that operates on field data to produce new data.
- **SuperGlue,** *NASA Ames.* SuperGlue[HR92] aims to emphasize extensibility (more than ease of use) for the rapid prototyping of new visualization methods by providing a programming environment based on the interpreted language Scheme. It is especially tailored for computational fluid dynamics needs.

Modular Visualization Environments

- **Advanced Visualization System (AVS),** *AVS Inc.* This is the earliest, general purpose, data flow based visualization system that has a wide user community[Ups89, Wet90b, AVS92, Bel93]. The architecture consists of five layers. At the bottom is the *system interface* layer that supports native graphics libraries of the platforms AVS runs on. On top of this is a *system independent* layer that uses a 3D rendering abstraction to make the system easier to port. The *renderer* layer includes renderers such as the image renderer and the geometry renderer. The *executive* layer includes a command language interpreter, the flow executive and the modules. At the top is the *user* layer that includes the network editor and various viewers such as the geometry and image viewers.

From a user's point of view, AVS consists of interactive applications that are made up of viewers and the network editor. These can run as standalone turnkey applications. The user can drive AVS either through graphical user interfaces or the command language. Graphically, the user builds applications by connecting modules in the network editor. The editor ensures data compatibility through strong typing.

Modules are the fundamental computation units that process inputs and generate outputs. There are four categories of modules: data sources, data filters, data mappers and renderers. Data sources perform the data import function, converting input formats to the internal AVS data types. Data filters operate on a data type and change it. Example filters are ones that crop, transpose or do a histogram equalization on an image. Data mappers produce renderable output from an input field. Examples include isosurface extractors and alpha-blending volume visualizers. Renderers manipulate renderable objects and make them visible. In addition to subroutine modules which execute whenever inputs or parameters change, there are coroutine modules that execute independently, obtaining inputs from AVS and sending outputs to AVS whenever it wants. The latter can be used for such things as integration of AVS with a simulation and the control of the flow executive for keyframe animation.

The flow executive of AVS is the component that schedules the execution of the modules. It also supervises data flow between modules, keeping track of where data is to be sent and by what method to send data. It is based on the data flow architecture such that the operations (modules) are enabled if and only if the required input values have been computed. The modules "consume" input values and "produce" output values. Hence, the only sequencing constraints are those imposed by data dependencies.

Some significant features of AVS are the following:

- Multiple modules can exist in a single process so the number of processes launched by an AVS network can be lowered to as few as one.
- As long as a module has no input dependency upon other modules that would be executing at the same time, modules will execute in parallel.
- Modules can execute on remote machines. AVS uses XDR for remote module communication.
- Shared memory is used for data passing between modules in different executables.
- AVS allows upstream data flow suitable for loops and conditionals in the network.
- AVS can be used to build customized applications.

- **apE,** *TaraVisual Inc.* Another early package, apE was originally developed by Ohio State University and was public domain until it was licensed to TaraVisual for marketing[Dye90, Wet90a]. It is a general purpose, data flow visualization system. The work area where the visual program is constructed is known as the *wrench.* The connections of modules in apE are not strongly typed and the process of connecting is less smooth than AVS, requiring the user to occasionally type in text. Modules are separate processes as in AVS, however, the processes are not forked when they are dragged into wrench. Execution of the program begins by a specific start button when all the processes are created. apE uses a data format called *flux* which not only describes the data but also the program pipelines, images and object properties. While it supports different grid types, the modules provided do not support all types. A module called *rezone* maps variables from one grid type to another. Distributed applications are possible in apE. There is a post-processor for image processing.

- **Iris Explorer,** *Silicon Graphics Inc.* Iris Explorer[Slo92, Edw93, Bel93] is another general purpose, data flow visualization system that has been bundled with Silicon Graphics machines. At the present time, it only runs on SGI machines but version 3.0 will be unbundled and ported to other machines. The architecture consists of five layers. At the bottom is the *system* layer that includes the standards the system is based on, e.g. X, Motif, Unix, GL. The *programmatic support layer* is a collection of libraries that constitute the API for module writers. The *module* layer is the collection of modules that handle the standard visualization tasks. The *map* layer contains the visual programs that have been constructed from the modules and the final *application* layer contains standalone application networks for standard disciplines.

Explorer consists of three programs.

- The *Map Editor* is the visual tool for constructing a visual program from modules. Each module's control widgets can appear in a separate window, can be hidden or can be seen live but minimized with the module's icon. Modules can be grouped together to save screen space and selected widgets can appear on its single control panel. Hence, at the extreme, a map can be grouped into a single module. Such a module can also be wrapped into a standalone application that does not require the map editor to run. A *parameter function* editor allows expression evaluation so that the input parameter of a downstream module can be controlled by the parameters of a combination of upstream modules.
- The *Module Builder* is an interactive, visual tool that lets expert users integrate modules to the system by transforming C, C++ and F77 code to Explorer modules. It includes visual tools to define the input and output ports of the module, to connect the ports to the functions arguments and to create and design the control panel.
- The *Data Scribe* is another visual tool that allows the import and export of data formats. The user can create an Explorer module that reads a data file and extracts relevant fields to construct a native data type that can be used in a map.

The execution of a map in Iris Explorer is based upon a distributed, decentralized data flow execution model. There is a single global server (GC) that manages the communication between modules on different hosts. There are also local communication servers (LC) on each machine. User interaction is conveyed to the GC which delivers it to the LC which forwards it to the module. Modules, which are processes, communicate directly with each other. Those on shared memory machines use named pipes while those on different machines transfer data through sockets. A module fires when the required inputs are present at its ports. To avoid multiple firings in a map, data tagging is used and the network topology is broadcast to all the modules by the GC.

Explorer offers a module prototyping facility through an interpretive language called *shape*. A scripting language called *Skm* allows scripting as well as a command language interface to the user.

- **IBM Visualization Data Explorer,** *IBM Corp.* Data Explorer[LAC$^+$92, Bel93] has a client-server open system architecture that uses standard protocols and systems for portability. It consists of two components, the user interface (the client) and the executive (server) running as two separate processes communicating via Unix sockets. The user interface includes the visual program editor, the control panel and the image and help windows. The server includes the executive, the modules and the data management API.

  The user interface provides a visual programming interface similar to Iris Explorer and AVS. The visual program is translated into a script language and sent to the executive for interpretation. The executive is the component of the system that manages the execution of the modules. The scripting language is also available to the user for writing scripts. The design differs from AVS and Explorer in that the modules are invoked not as separate processes but as function calls. The data management layer is the portion of the programming interface that provide modules with access to the data model. The data model is unique among the packages in that it is based on the mathematics of fiber bundles.

  The data flow execution model of Data Explorer imposes the constraint that modules possess pure function semantics in that outputs are based on the inputs and not on some state that is the result of a previous execution. Also, modules treat their data as read-only. A side benefit of pure function semantics is that caching of intermediate results is possible. Distributed processing is possible allowing modules or groups of modules to be executed on different machines permitting enhanced resource utilization.

Support for coarse-grained shared memory parallelism is provided but makes module writing more difficult. Data Explorer uses explicit data partitioning and a simple fork-join model to make this task easier.

2.2 Spray Rendering

Spray rendering was proposed by Pang and Smith[PS93a, PS93b] as a new framework for doing scientific visualization. In this framework it was proposed to have a shelf of metaphorical spray cans containing smart particles (*sparts*) designed to look for features in the data set and manifest themselves as renderable objects. Users would select one of these cans and spray the data set, select another to obtain a different effect and through an iterative process, achieve a certain visualization. The original ideas were high level concepts of what could be accomplished with sparts.

The main idea of spray rendering was to combine particle systems and behavioral animation. Particle systems are an area of computer graphics that have been used to model natural phenomena such as fire, water and grass which are difficult to model by traditional methods. In this technique, objects are represented by a collection of particles as opposed to surface elements. These dynamic entities change form with time as new ones are born and old ones die. Stochastic processes are used to affect the shape and form resulting in non-deterministic objects. The particles have attributes such as size, color and shape, and the system as a whole has parameters governing its form. Reeves used this technique to model fire, explosions, fireworks and grass[Ree83]. Behavioral animation was developed to capture group behavior in animations. Rather than account for each body in a group individually, the individuals in the group are required to behave according to some rule based constraints set for the whole group. The constraints, such as collision avoidance and maintained average speed, endow the group as a whole with a behavior. This mimics the behavior of a school of fish or a flock of birds rather well[Rey87, Amk89].

In the light of the above techniques and an object oriented preliminary design, sparts were to have a state consisting of certain attributes such as age and appearance, and certain methods to affect that state such as: target function, direction function, death function etc. They would have a local neighborhood that they would process, and this neighborhood would be updated as the spart traveled. Sparts were envisaged to work individually or cooperatively. Spart to spart interactions were to take place through the deposition of *markers*. These were distinct from visualization objects in that they could not be rendered but would contain communication information. Sparts would also be organized into groups and hierarchies.

2.3 How *Mix&Match* relates to other environments and spray rendering

The extensible, general purpose scientific visualization packages have become the *de facto* standard. These recognize the diversity of visualization needs and provide room for growth whereas the turnkey systems are constrained by the number of tools they come with. The extensible systems have also reached a consensus on providing a data-flow oriented visual programming interface for composition. The popularity of these systems can be attributed to their promotion of software sharing and their extensibility and relative ease of use. Scientists or visualization programmers can develop modules for their own needs and share them with other scientists.

There are similarities as well as differences between *Mix&Match* and the MVEs. The similarities are:

- *Extensibility.* This is a major shared characteristic and is a determining factor in allowing users to meet their own needs.
- *Modularity.* Users extend the system by adding modules. These modules can be combined to accomplish a visualization task at runtime.

- *Ease of Use.* Users are presented with a graphical interface to build the networks (visual programming).

At the same time, there are very important differences.

- *Modules.* The granularity of the modules in MVEs are coarse, whereas in *Mix&Match* they are fine. Coarse modules operate on the whole data set, while components in *Mix&Match* act upon data at a current locality. The components of *Mix&Match* are, in general, simple and small and accomplish a very specific task. Finer granularity allows greater flexibility in composition and their simplicity facilitates easier system extension. This, in turn, results in a richer repertoire of techniques and encourages experimentation and exploration.
- *Networks.* The networks in MVEs define a program the result of which is the visualization. Those in *Mix&Match* define a program that determines the overall behavior of a spart.
- *Execution Model.* MVE networks use a data flow execution model. Modules execute when new data arrive at their ports. The composition in *Mix&Match* constitutes a program that gets executed at each location the spart occupies during its lifetime. In effect, rather than data flowing through modules, it is the modules that flow through the data.
- *Visualization Process.* In MVEs, the visualization is the result of the network program. If a parameter of a module changes, the program re-executes and a new result is obtained. In *Mix&Match*, the composition defines a spart and the spray can containing the spart becomes a tool. Visualization is an iterative, interactive process of applying this tool. Complex visualizations can be obtained by multiple uses of multiple spray cans.
- *Network complexity.* Related to the previous item, network complexity increases in MVEs if more complex visualizations are desired. The networks in *Mix&Match* are, in general, simple and, since multiple networks can be used multiple times, the complexity of the end visualization does not affect the complexity of the network.

- *Direct Interaction.* Very little user interaction is supported in MVEs. The use of the spray can metaphor in *Mix&Match* permits direct interaction with the data set, emphasizing the exploratory aspect of the visualization process. System level mode settings allow the same composition to be used in different interactions.

Mix&Match can also be compared to tool-based turnkey systems. It is tool-based since each spray can containing a visualization technique is essentially a tool, and multiple instances can be used on the data multiple times. Unlike turnkey systems, however, new tools can be defined and tool definitions can be changed at runtime. New components can also be added which makes it an extensible system.

Although this thesis uses the metaphors of spray cans and sparts, the design and implementation of spray rendering in this work is less ambitious than originally envisaged[PS93a, PS93b]. As will be elaborated in later chapters, the emphasis in *Mix&Match* is on the ability to interactively define the overall behavior of a spart and different interaction techniques. In *Mix&Match*, sparts are independent and do not communicate.

3. Scientific Visualization: Techniques

In designing and developing the *Mix&Match* scientific visualization environment, some of the popular techniques were implemented by adapting them to work in the spray rendering model and decomposing them into basic components. In this chapter, a context for these techniques is provided by giving a brief survey of scientific visualization techniques in general. The chapter starts with a definition of scientific data and a classification of the more important volume visualization techniques. This classification leads to a discussion of which techniques are more amenable to be adapted to the spray rendering model.

3.1 Scientific Data

In the most general case, a simple definition[A$^+$92] treats scientific data abstractly as a mapping between an n-dimensional space of independent variables $\mathbf{x}$ and an m-dimensional space of dependent variables $\mathbf{y}$. This mapping can be represented by the following matrix:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_m \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, x_3, \ldots, x_n) \\ f_2(x_1, x_2, x_3, \ldots, x_n) \\ f_3(x_1, x_2, x_3, \ldots, x_n) \\ . \\ . \\ . \\ f_m(x_1, x_2, x_3, \ldots, x_n) \end{bmatrix}$$

This high level abstract definition hides the diversity of scientific data generated. Some attributes that can be used to categorize data are the following[Tre93]:

- **Physical Data Type Primitives:** The data can be stored on a medium in many ways, (e.g. byte, int, float, ...).
- **Dimensionality:** (e.g. spatial, temporal, spectral, ...). Scientific data may be spatially coherent, i.e. the independent variables are spatial dimensions as in CT scan

data or they may be spatially non-coherent as in census data. The size, shape and organization of the independent variables are also important.

- **Rank:** This refers to the number of values per element (e.g. scalar, vector, tensor, ...).
- **Mesh description:** Often, the elements have a mapping to some physical domain or coordinate system and the mesh description identifies the size, shape and organization of this mapping (e.g. regular, irregular, curvilinear, ...).
- **Aggregation:** This refers to a collection or organization of a set of functional values (e.g. hierarchies, groups, series, ...).

Up until a few years ago, scientific visualization programs and packages were domain specific, often developed in-house. Each discipline's needs were different and little data sharing was taking place. For this reason, a multitude of data formats evolved. To remedy this problem and encourage the sharing of data, attempts at standardization took place in the late eighties. CDF ([Gou88]), netCDF ([RD90]) and HDF ([Nat89]) have found wide usage among certain disciplines, but many of the visualization packages still use their own formats.

Another motivating force for an abstract data model, not just a standard format, is the need to manage enormous amounts of data generated through simulation and observation by the scientific community. The goal is to link scientific visualization and DBMS technologies[SCN$^+$93, KASS93]. An interesting data model is one proposed by Butler and Pendley[BP89], a model based on the mathematics of fiber bundles which was extended by Haber *et al.* [HLC91] to incorporate localized, piecewise field description. Their model is especially suitable for representing continuum fields although it can also represent random sample points and ball-and-stick molecular models.

3.2 A Classification

A classification of the main volume visualization techniques (ignoring the many variations) can be helpful to put things into perspective. It also provides insight as to which techniques are more suitable to the proposed framework.

Treinish lists general visualization techniques based on the dimensionality and the rank (see section 3.1) of the data[Tre93]. The table includes example data types and the techniques are classified as either discrete or continuous. Upson provides a two dimensional array where one axis represents the dimensionality of the computational domain, the other the dimensionality of the visual representation and the techniques appear as elements in the array[Ups91]. Another classification by Hesselink *et al.* also adds a third attribute, the *information level*, which indicates whether the information shown at a certain point refers only to the elementary data at that point, whether it refers to some local neighborhood or whether it is global to the whole data set[HPvW94].

Table 3.1 lists some 3D techniques in a manner that is a combination of the above classifications. The *Geometry* column of the table refers to the dimensionality of the geometric primitive used to represent the data. These could be points (0), lines (1), polygons (2) or volumes (3). Also included are glyphs (G) and pixels (P) in this column, because such techniques do not fit the other descriptions. For instance, the *cuberilles* technique places opaque cubes at places satisfying a given threshold (see section 3.4.2). This is really a glyph representation rather than a volumetric one. Ray casting direct volume rendering algorithms work in image space and accumulate pixel values and hence do not have a geometric representation, although the principle behind their operation is volumetric.

The *Rank* column represents the dimensionality of the data. Scalar fields (rank 0) are single-valued functions, vectors (rank 0) of dimension n are n-valued functions and tensors (rank 2) in an n-dimensional space are n x n valued. The *Information* column refers to the information level shown at a point as discussed above (E=elementary, L=local, G=global). The *Discrete* column classifies the technique as being either a discrete (D) or a continuous approach (C). In 3D, one could say that any geometric representation that is not volumetric would be discrete. For instance, an iso-surface at a certain threshold is only showing a portion of the data set. On the other hand, it can be regarded as a continuous technique since the polygons generated in cells have continuity as opposed to simple glyphs placed at

Technique	*Geometry*	*Rank*	*Information*	*Discrete*	*Objects*
Scatter plots	0	0	E	D	Y
Dot Surfaces	0	0	E	D	Y
Wireframe Surface	1	0	E	D	Y
Contours on slice	1	0	E	D	Y
Streamlines	1	1	E	C	Y
Hyper Streamlines	1	2	E	C	Y
Pseudo-colored slice	2	0	E	C	Y
Iso-surfaces	2	0	E	C	Y
Stream Ribbons	2	1	E	C	Y
Cuberilles	G	0	E	D	Y
Hedgehogs	G	1	E	D	Y
Tensor Probe	G	2	L	D	Y
Projection VR	P	0	E	C	N
Ray Casting VR	P	0	E	C	N
Hierarchical Splatting VR	2	0	E	D,C	Y
Vector field topology	1,2	1	G	C	Y

Table 3.1: A classification of some visualization techniques. *Geometry* and *Rank* refer to the dimensionality of the geometric primitives and the order of data, *Information* refers to the locality of information represented by a point, *Discrete* refers to whether the technique is discrete or continuous and the *Objects* column refers to whether visualization objects can be produced locally.

sample locations. Such techniques are classified here as continuous to capture this nuance.

The final column indicates whether visualization objects can be generated locally and sent to a renderer. Since the components in *Mix&Match* operate locally, those that produce visualization objects have to have this property. For instance, a coarse grained streamline algorithm would accumulate the vertices corresponding to each new location during the vector integration phase and output a line-segment set based on these vertices. A fine-grained approach would need to output a line segment at each new location.

Mix&Match uses spray rendering to model the components. The smart particles delivered from the spray can travel in the data set and produce renderable visualization objects. Because of the discrete nature of this process, discrete techniques that can generate geometric objects map most naturally to *Mix&Match.* However, continuous techniques that can be decomposed into components that can generate portions locally can also be implemented. For instance, the marching cubes iso-surface extraction technique is a continuous technique, but the polygons making up the surface are generated in each cell. A smart particle traveling in the data set can generate the polygons of the particular cell it is in.

The volume rendering techniques are more difficult to handle since they do not produce geometry, but result in accumulated values for pixels. If volume rendering is to be combined with geometry, the task is even harder. A volume rendering technique that may be more amenable for implementation in *Mix&Match* is the *splatting* technique that relies on geometry (see section 3.4.3). A hybrid software renderer that mixes geometry and volumetric data may be required to handle this.

The following sections gives a brief survey of two and three dimensional visualization techniques.

3.3 Two Dimensional Visualization

A very common scientific data type is a scalar value, such as temperature, that varies over a two dimensional region, i.e. a bivariate function $F(x, y)$. The techniques to visualize such data are very familiar to even the non-scientist because of their omnipresence. Weather maps in forecasts on TV or in newspapers provide a daily dose of these.

The most common technique is *contouring*. Contour lines are drawn over the 2D region that represent the locus of points that have the same given value. There are two main algorithms: producing all the contour lines within each cell so that contours are produced in a piecemeal fashion, or following each contour to its conclusion. Sabin gives a survey of contouring methods that also deal with scattered data[Sab86]. Instead of drawing curves, one can map the values to color and use color blended regions, so called *pseudo-color* contour maps.

Another popular method, sometimes referred to as *rubber-sheeting*, is to render a surface of the bivariate function by projecting the value of the function as a distance from the point (x, y) in the planar domain. This is especially suitable for terrain data where the value of the function is actually in the third dimension.

3.4 Volume Visualization

The term volume visualization refers to the process of obtaining a visual representation of a collection of sample values located in three dimensional space. In this case, the underlying function $F(x, y, z)$ is trivariate. If connected by a grid, sub-volumes formed by neighboring sample points are called *cells* in which the underlying function is assumed to vary continuously. Some algorithms refer to the sample values as *voxels* where it is assumed that the sub-volume surrounding the sample point is constant-valued.

A taxonomy of the connectivity appears in [SK90] (although this is by no means standard):

- **regular:** Cells are identical parallelepipeds (bricks). They may have equal distances along each axis in which case they are cubical cells.
- **rectilinear:** Cells are no longer identical but they are still bricks and axis-aligned.
- **structured:** Also known as *curvilinear*, cells are no longer bricks but have been warped by a transformation. They are still made up of eight points; however, faces may not even be planar any more.
- **block structured:** Several structured grids may together make up a block structured grid.
- **unstructured:** Cells in this type of grid may be of a variety of shapes. The connectivity is supplied.
- **hybrid:** Any combination of the above may make a hybrid grid.

3.4.1 Slicing, Probing and Carving

A common technique of volume visualization is to slice the volume orthogonally or at an arbitrary orientation and use a 2D technique such as contouring or pseudo-colored slices. When these slices are animated across the volume, the motion provides extra visual cues[Smi87]. A related technique uses geometric objects as probes to interactively examine the data values on the surface of the probes[SK90]. One can also carve out sections of the data set to either remove uninteresting portions or to reveal hidden areas[FZY84, FGR85]. These can be done by boolean set operations as in constructive solid geometry. Any of the volume visualization techniques can then be used to view the remaining volume of interest.

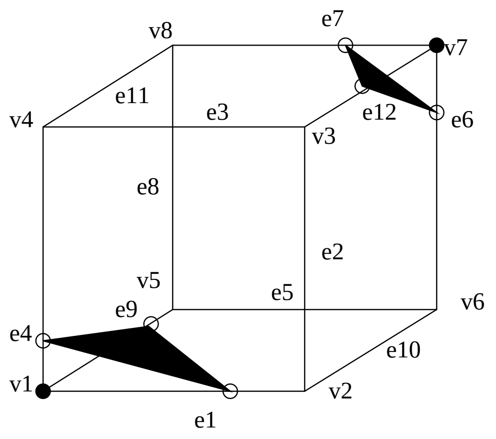
3.4.2 Geometry-based Volumetric Techniques

Geometry-based volumetric techniques produce geometric primitives that are rendered. Some algorithms use a binary voxel classification. Herman and Liu[HL79] first described the *cuberille* technique, which thresholded volume data to yield a binary array of ones and zeros. The resulting volume was displayed by treating ones as opaque cubes. Frieder *et al.* [FGR85] and Gordon and Reynolds[GR85] improved upon the algorithm by processing the voxels in a single pass. Meagher proposed the use of octrees for speeding up the process[Mea82]. If local gradient shading is applied instead of a binary representation substantial improvements to image quality can be obtained[HB86, Gol86, SSW86, TS87].

Three dimensional representations can be extracted from two dimensional contours which can be obtained through edge tracking[FKU77]. These techniques evolved into extracting surfaces directly from the volume data by specifying a threshold value. Particularly popular techniques for extracting so-called *iso-surfaces* is the *marching cubes*[LC87] and the *dividing cubes*[CLL$^+$88] algorithms.

In the marching cubes algorithm, one walks the cells in a volume and marks the vertices of a cell as either ones or zeros depending on whether the value at that vertex is above or below a given threshold. These bits encode a tag for the cell which is used as an index into a case table (figure 3.1). The case table enumerates all possible cases and is used as a look-up table to define the polygons that exist for a given case. Edge intersections are computed by interpolation and a central differences formula estimates the gradients used for shading. In the dividing cubes algorithm, the cells are subdivided until their projection is a single pixel. The geometric primitives used in this case are points.

The original marching cubes algorithm suffers from ambiguous cases and can lead to holes in surfaces where none should be present. Solutions to this problem have been



v8	v7	v6	v5	v4	v3	v2	v1
0	1	0	0	0	0	0	1

Figure 3.1: The figure illustrates a marching cubes case where vertices v1 and v7 are above while the others are below a given threshold causing edge intersections e1,e4,e9 and e6,e7,e12. The index and the triangles produced are also shown.

proposed by [WVG90b, NH91, Mat94]. The marching cubes algorithm also produces a large number of geometric primitives. To increase the efficiency, Wilhelms and Van Gelder[WVG90a] proposed a hierarchical approach while [Tur92, SZL92, MSS94] have proposed approaches to reduce the number of the primitives. Other related techniques for extracting and rendering surfaces use implicit surface methods[Bli82a, WMW86, Blo88].

3.4.3 Direct Volume Rendering

The term "direct" in Direct Volume Rendering emphasizes the distinguishing characteristic of the technique: no intermediate geometric primitives are produced, as is the case in isosurface extraction. The technique has increasingly become popular, despite its computational cost, because each and every sample value in the data set contributes to the image. The mappings from data to visual parameters are extremely flexible, resulting in a variety of images emphasizing different aspects of the data set. It is also more appropriate for certain volumes where the sample values represent amorphous, cloud-like phenomena where surfaces do not really mean much.

The essence of the technique is that for each pixel, a color value is accumulated based on the contributions of color and opacity from the sample values. The algorithms differ in the order that they proceed and in the way that they map and accumulate the color and opacity values. The algorithms can basically be classified into *object order* and *image order* algorithms. In the former, one starts from object space and calculates the contributions of the sub-volumes to the affected pixels. In the latter, for each pixel in image space, one accumulates the contributions of the sub-volumes affecting it.

Image Order Algorithms

Image order algorithms are ray-casting techniques[Lev88, Sab88, UK88]. This is distinct from ray-tracing since rays are not reflected from objects. Instead, rays are cast from each pixel into the data set in object space which they enter and exit in a straight line (figure 3.2). Samples are then taken along the ray either by taking equal steps or by going from cell face to cell face. In the former case, values of samples that fall in a cell are normally obtained by trilinear interpolation which can be defined as:

$$f(x, y, z) = a + bx + cy + dz + exy + fxz + gyz + hxyz$$

The coefficients $a \ldots h$ can be calculated by evaluating the above equation at the eight corners of the cell. Higher order interpolations are also possible but are not normally used due to their cost. With ray/cell face intersections, bilinear interpolation is used to calculate the sample value. For each of the sample values, a color and opacity value is calculated. These are then composited in front-to-back or back-to-front order to give the final pixel color[PD84]. For instance, the front-to-back color and opacity compositions are done according to the formulas:

$$\begin{aligned} C_{composite} &= C_{front} \times \alpha_{front} + C_{back} \times \alpha_{back} \times (1 - \alpha_{front}) \\ \alpha_{composite} &= \alpha_{front} + (1 - \alpha_{front} \times \alpha_{back}) \end{aligned}$$

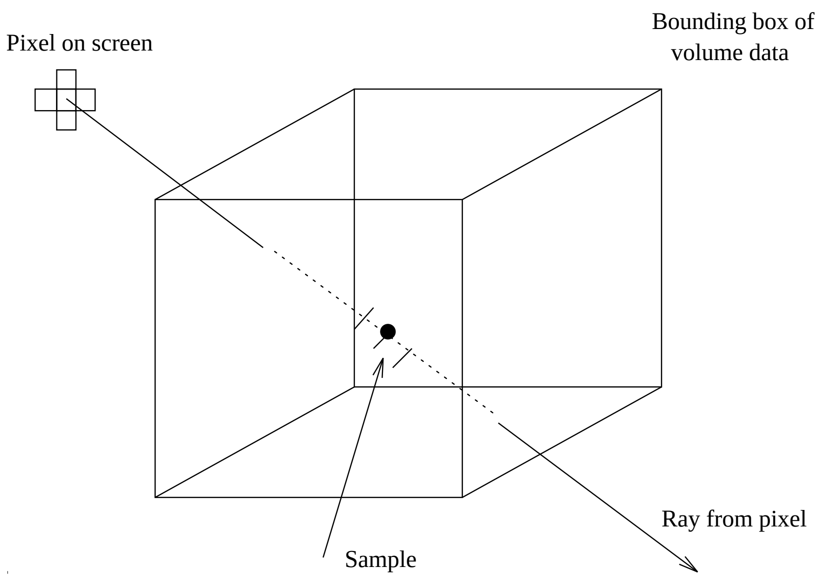


Figure 3.2: Rays are cast from pixels in image space into the volume in object space and samples are taken along the ray in image order volume rendering algorithms.

where C is the color and α is the opacity.

The ray-casting algorithms differ in the details of the model used for mapping to visual parameters. Sabella[Sab88] uses a simplified light scattering model that had been used for the image synthesis of natural phenomena such as clouds[Bli82b, KH84, Max86]. In his *density emitter* model, the volume is assumed to consist of light-emitting particles but instead of modeling the particles individually, he considers their density and derives a ray integral involving an exponential for the attenuated intensity along a ray. This integral equation is then approximated by a discrete sum of products equation.

Upson and Keeler[UK88] use independent color and opacity transfer functions for mapping the scalar values. They use finite differences at the nodes to calculate the normals used in shading. The integral is approximated as a discrete summation.

Levoy[Lev88] calculates colors and opacities from the scalar values through shading and classification formulae respectively and uses trilinear interpolation on these volumes for samples that fall in cells. The shading model used is the standard Phong model. For classification, opacity is made to be a function of the local gradient. These functions depend on whether one is trying to extract value contours or region boundary surfaces.

Other ray-casting approaches that have appeared in the literature are as follows: Tuy[TT84] used ray-casting to render binary voxels, Montani[MS90] rendered constant valued voxels using the *sticks* representation, Novins[NSG90] used a slab based method to achieve perspective projection, Krueger[Kru90] developed an elaborate and flexible transport theory model, Garity[Gar90], Challinger[Cha90], Wilhelms[WCA$^+$90] raytraced irregular volume data and Yagel[YK92] used a template-based ray-casting method for rendering constant valued voxels. More recently, Stander and Hart have used a Lipschitz method for accelerating a ray-casting volume renderer[SH94].

Object Order Algorithms

Object-order algorithms are projection techniques where the main loop of the algorithm proceeds in object space. Researchers have taken two approaches in this case. Most break down the volume into sub-volumes and scan convert the front and back faces of the sub-volumes in front-to-back or back-to-front order[UK88, MHC90, ST90, WVG91]. A different approach called *splatting* composites the footprints of each node in the volume[Wes89, Wes90].

Upson and Keeler's[UK88] *V-buffer* algorithm determines a bounding box for each cell in front-to-back order. The bounding box is clipped to scanlines to produce pixel runs. Each scanline can be broken up into five spans depending on the polygon produced by the cutting plane of the scanline. The calculations in the spans are vectorizable.

Max *et al.* [MHC90] and Shirley and Tuchman[ST90] handle not just regular grids but curvilinear ones as well. They break down the cells into convex polyhedra (tetrahedra in the case of[ST90]), sort them in depth order and scan convert them. Max *et al.* use an assumption to provide an analytical solution to the ray integral equation. Wilhelms and Van Gelder[WVG91, GW93] employ a similar projection technique for rectilinear and curvilinear volumes. All of them provide options for using hardware interpolation of color and opacity values across polygons to achieve greater speed at the cost of some accuracy.

A different approach was taken by Westover[Wes89, Wes90]. He used a reconstruction kernel (a Gaussian) and proceeded from each node in the grid to calculate the contribution its footprint made on the affected pixels, a process he termed *splatting.* All the algorithms use the same compositing scheme for accumulation of the contributions to affected pixels[PD84].

3.5 Multiparameter and Vector Visualization

Multiparameter visualization, as used in this document, will mean more than one dependent variable associated with the independent variables, and is restricted to spatially coherent techniques (the independent variables are spatial). Vector visualization is defined as having a vector quantity, such as velocity, at each data position in 2D or 3D space.

Spatially coherent multiple parameter data is quite common in science. For instance, over a 2D region, one might measure multiple physical quantities such as temperature and pressure. Scientists are then interested not only in a single parameter's variation over this region but also in the relationships among the multiple parameters. One way to visualize these is to use a visualization technique on each of the parameters separately and display them in multiple windows. The scientist is then left with the task of visually comparing the images in an effort to determine relationships between the different parameters. This does not support the determination of more subtle relationships among the parameters. As discussed next, some researchers have attempted to alleviate this problem by using visualization techniques that use a single, integrated display.

In 2D, *iconographic* displays have been used to visualize multiparameter data. The icons consist of a number of pixels and are coded both in terms of color and geometry. Different parameters of the data set govern different colors and features of the geometry of the icon[Lev91]. Crawfis and Allison use an interpreted programming environment to synthesize textures and raster images which can be composited together[CA91]. The textures and images can be obtained by operating on different parameters and integrated to reveal relationships.

Foley and Lane describe various techniques that can be incorporated into a single image to visualize 3D multiparameter data sets[FL91]. The techniques involve defining a surface or a geometric object in the volume and operating on that volume. By combining the operations in a single image, different parameters can be related to one another.

Two dimensional vector visualization is also quite common. The most widespread method is to draw arrows at the data points where the length of an arrow is proportional to the magnitude of the vector and its direction indicates the direction of the vector. Often, drawing one vector glyph at each point creates a crowded image so vector glyphs are drawn at positions which have been subsampled from the original data set. Alternatively, contour maps or streamlines can be used. Color can provide additional cues. Texture has also been used in visualizing 2D vector fields[vW91].

Three dimensional vector visualization has been extensively studied under the name *flow visualization*, and many techniques exist. One approach is to reduce the dimensionality by visualizing the field on a cutting plane or an arbitrary object's surface. Any of the two dimensional methods discussed above can now be applied to this surface. Another technique is to calculate a scalar value from the vector field, such as helicity density, and visualize the scalar value using any of the volume visualization methods.

Techniques have also been developed to visualize the vector field directly. The simplest approach, sometimes called the *hedgehog* method, is to display 3D arrow glyphs at the data points, again subsampling to reduce image clutter. This does not work as well as in 2D since the arrows are projected onto the screen and it is more difficult to get the magnitude and direction information from the projection. Particle-based techniques offer better insight. These techniques have their parallels in the laboratory setting. Fluid dynamics scientists release dyes into liquids and smoke into air flows in order to study fluid flow.

The simplest method of flow visualization is to follow the motion of massless particles released into the flow field. The particle is drawn at the release point (e.g as a sphere), and its next position is calculated using the vector field. The calculation is a simple integration where one assumes that the vector is tangent to the path. If $\mathbf{r}(t)$ is the position vector of a point on the path where t is time, the velocity vector $\mathbf{v}(\mathbf{r})$ is given by

$$\frac{d\mathbf{r}}{\mathrm{dt}} = \mathbf{v}(\mathbf{r})$$

and the integral of this gives

$$\mathbf{r}(t) = \mathbf{r}(0) + \int_0^t \mathbf{v}(\mathbf{r}(t))dt$$

The particle can be redrawn at each new position so that one can see its motion through the field (*particle advection*). Instead of drawing the moving particles themselves, one might merely draw its continuous path. If this path consists of line segments, one gets so-called *streamlines.* The line segments can be colored according to the magnitude of the vector to give extra information. Issues such as the integration method, vector interpolation and step size adapting have appeared in the literature [MP88, Bun88, YP88, EOR89]. More recently, Kenright and Mallinson have proposed a new approach for tracking streamlines where streamlines are considered to be the intersection of two stream surfaces[KM92].

In 3D, one can get an improved perception if, instead of drawing lines, one draws 3D shapes such as cylinders. This results in *stream tubes* which can be shaded using lighting models[Dic89, HD91]. Stream tubes can also have the advantage of showing local expansion of the flow field if their circular cross-sectional areas are based upon the local crossflow divergence.

Stream lines or tubes only give the path. One cannot obtain rotation information from these techniques. To alleviate this problem, *stream ribbons* have been used. In this case, the path of two particles are traced and polygons are generated from their joint path[Bel87, Vol89]. Since flow fields diverge, ribbons may need to be adaptively split to get better polygonal representations. Hultquist has proposed an advancing front method that achieves this more efficiently[Hul92]. Another method that has been used to capture not only the rotation but also the strain and shear (or angular deformation) is the *stream polygon* method proposed by Schroeder *et al.* [SVL91]. In this method, local deformations affect the shape of a polygon that moves along a stream line such that its orientation is normal to the local vector. When the vertices of these polygons are joined, one gets a non-cylindrical tube with local deformations.

Van Wijk creates textures by *spot noise* to visualize scalar and vector fields on surfaces[vW91]. By varying the parameters of the spots, such as shape and size, different textures are obtained that best fit the problem. Van Wijk also uses *surface particles* to visualize flow fields[vW92]. Particles from a source are periodically released into the flow field and form a textured surface when rendered. By varying the shape of the source of the particles, he obtains different visualization techniques such as streamlines, tubes and ribbons. He provides a detailed method for rendering shaded particles.

Crawfis and Max[CM92] employ a splatting technique for the direct volume visualization of 3D vector fields. They have developed a filter which is used to sweep through a volume in back-to-front order. The filter deposits anti-aliased lines as it passes through the volume. The technique is used to volume render both a vector and a scalar field in the same image.

More recently, Max *et al.* used a volumetric equivalent of stream lines to volume render vector fields[MBC93]. Leeuw and van Wijk map scalar, vector and tensor values to a 3-D probe for interactive local flow field visualization[dLvW93]. Van Wijk also used implicit surface representation to construct implicit stream surfaces of flow fields[vW93]. Crawfis and Max extended the splatting technique to include textures for vector field visualization[CM93]. Cabral and Leedom introduced a novel technique which uses linear and curvilinear filters to locally blur textures along a vector field[CL93] which was extended by Forsell to visualize flow over curvilinear grid surfaces[For94].

4. Mix&Match: A Construction Kit

The environment presented in this dissertation blends the ideas of spray rendering with those of modular visualization environments. The metaphors of spray cans and smart particles (sparts) of spray rendering are used to model the visualization techniques and define various interactions with data sets. At the same time, the visual programming interface to program composition that characterizes modular visualization environments supports interactive composition of sparts. The result is *Mix&Match*, a flexible, modular and extensible environment which allows incremental visualizations through direct interactions. In this chapter, a high level view of the environment is presented, starting with the design goals.

4.1 Design Goals

For any software system there are basic software engineering properties that are desirable. Some of these properties are in conflict with one another and priorities and judicious tradeoffs are necessary. The design goals for *Mix&Match* were to achieve the following properties as much as possible.

- **Extensibility.** Extensibility relates to the ease of adding functionality to the system in existence. Although the system may have many features, it should allow easy growth to cater for needs not foreseen. The *Mix&Match* system has been designed so that the interface to the system of a component is generated automatically. The component writer implements the computational function using C code and the API provided, and specifies its inputs, outputs and other attributes graphically to integrate it into the system. The system has been designed to be modular to facilitate this extensibility.
- **Functionality.** The features provided initially as well as all the non-extensible parts of the system should be functional. Many visualization techniques have been implemented by redesigning and localizing them, resulting in many components to

construct sparts. Some predefined and composed sparts (see section 4.3) are also provided. The renderer allows a variety of renderings of the visualization objects (e.g. flat/Gouraud shading, point/wireframe/polygon drawing of primitives, lighting and viewing options).

- **Flexibility.** One of the advantages of having components of fine granularity is that the flexibility of component composition is enhanced. For instance, instead of a single iso-surface component, one can break it into two components: one that seeks to satisfy the target iso-value, and one that generates the surface if the condition is satisfied. This target function can also be used in another composition where some other visual behavior component is used. However, flexibility can be in conflict with efficiency. The finer the granularity, the more components to execute, and the larger the overhead. To compromise, the *Mix&Match* design limits the components to have granularity at the spart position level.
- **Efficiency.** The particle nature of spray rendering and the fine granularity of the components can be costly. Object compaction (see section 5.5) was designed to save memory and rendering time. A simple memory scheme for data transfer between components was designed so that components merely dereference pointers to read their inputs instead of communicating through files or pipes.
- **Ease of Use.** Since the target users are scientists more eager to do their science than to learn a software package, the environment has been designed to be as user friendly as possible. Graphical user interfaces and direct manipulation have been used throughout to aid in this process. The system aims to support three different levels of users: *Novices* merely load sparts "off the shelf". *Intermediate* users go a step further and use the components available to compose new sparts. A graphical spart editor allows intuitive visual programming for spart composition. Finally, the *expert* user can write C functions to add a new component to the system. A configuration manager presents a graphical user interface for this task as well and generates wrapper code to make the integration easier.

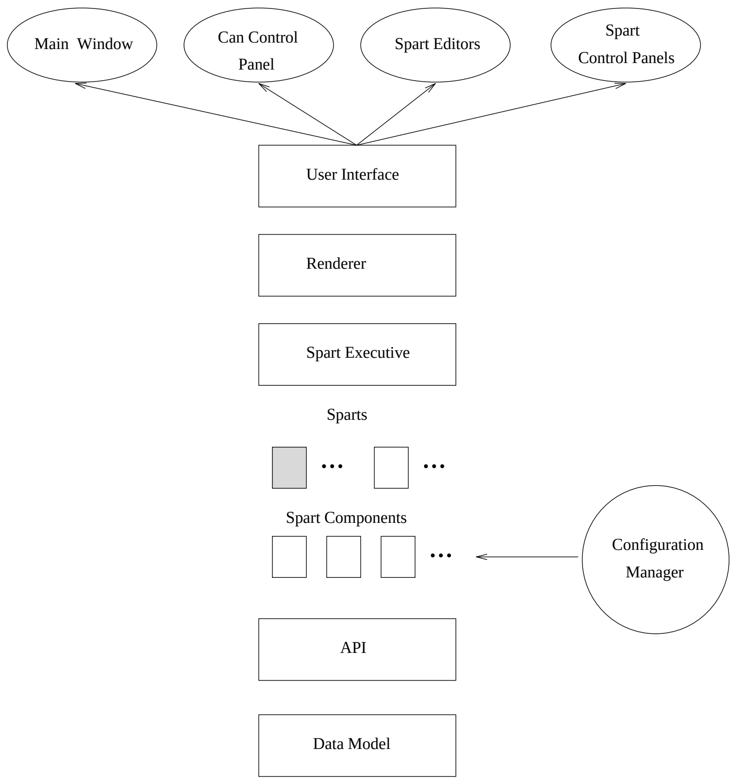


Figure 4.1: The organization of the system components.

4.2 *Mix&Match* Overview

Figure 4.1 shows how the components that make up the *Mix&Match* environment[1] are organized. At the bottom level is the *data model* used to define the data sets. An application programmer's interface (*API*) provides functions that operate on the data for the convenience of spart component writers. Next, there is an extensible collection of *sparts* and *spart components*. These form the heart of the visualization process since they implement the visualization techniques. Spart components can be composed to define an overall behavior for the spart. Predefined sparts already have their behavior defined and cannot be composed. Both spart components and predefined sparts are functions and

[1] The *Mix&Match* environment was developed as part of the REINAS project on meteorological and oceanographic data acquisition and visualization. What is described here is the analysis mode of the visualization program *Spray* in which users can visualize model simulation data.

are integrated into the system by linking. They are described to the system through a separate program called the *Configuration Manager*. This program presents a graphical user interface for component description and control panel design, and generates "wrapper code" for system integration so that the component writer can concentrate on writing the computational function. The *spart executive* handles the execution of the sparts while the *renderer* renders the scene (made up of the visual objects that have been generated by the sparts).

At the top of the system architecture is the *user interface*. The main interaction window presents the rendered scene and has a top-level menu bar. Users can interact directly with the window for view control and spray can selection and manipulation. The spray can control panel presents choices for can parameters and interaction modes. A textual and a graphical spart editor are both provided for the composition of sparts. Finally, most spart components will have a control panel to set certain parameters. When a can is created, these are collected into a single window.

Users of *Mix&Match* load in data sets called *streams*, create spray cans that contain particular sparts, and spray the data set with them to produce visualizations interactively. The process of visualization is summarized in figure 4.2. It is an iterative process and at any stage the result could be the desired visualization.

Users first load streams. They can then create a can containing a particular spart. The spart may be an existing spart composition or the user may edit an existing spart or may compose one from scratch. The various parameters of the can and the spart may then be selected and the can used on the data set interactively. This results in certain visualization objects which can be accumulated, hidden or deleted at any stage. By iterating on these various actions, the user obtains a visualization. The iterative process emphasizes the exploratory nature of this framework. Note that to facilitate a faster beginning to a visualization session, some of these operations can be specified from a startup file.

Figure 4.3 shows the top-level interface. On the left are three browsers that list the cans that have been created, the currently available sparts and the currently loaded streams. The

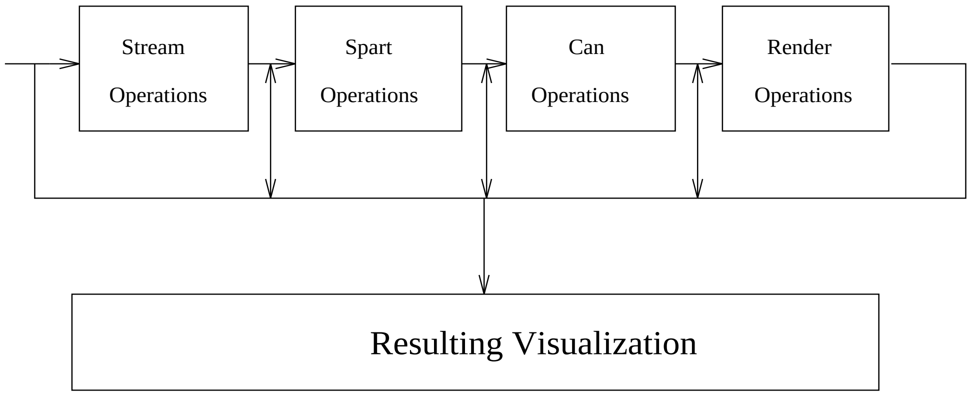


Figure 4.2: This figure illustrates the iterative process of obtaining a visualization in *Mix&Match*. Users operate on streams, sparts, spray cans and the rendering in the main event loop. It emphasizes the exploratory nature of visualization. At any stage, the user has a current visualization which could be saved as an image.

main graphics window shows the current state of the visualization and the smaller graphics window shows the world from the point of view of the current can. The various actions such as the loading of streams and the creation of cans take effect in response to selections from the menu bar. Users are able to interact with both the main window and the can window for such operations as the changing of view, can manipulation and spraying.

4.3 Smart Particles

A defining characteristic of spray rendering is the use of smart particles (*sparts*). These are launched from a metaphorical "spray can" and the particles actively look for features in the data set. Once these target conditions are satisfied, *abstract visualization objects (AVOs)* may be deposited which are subsequently rendered. A typical spart, then, has a life-time as depicted in figure 4.4. The target and visual behaviors take place at the current location of the spart. The spart then updates its position and determines if it is to continue or die.

Mix&Match allows sparts to be defined at run time by connecting together basic components. These sparts are called *Composed* sparts, and the components are organized into

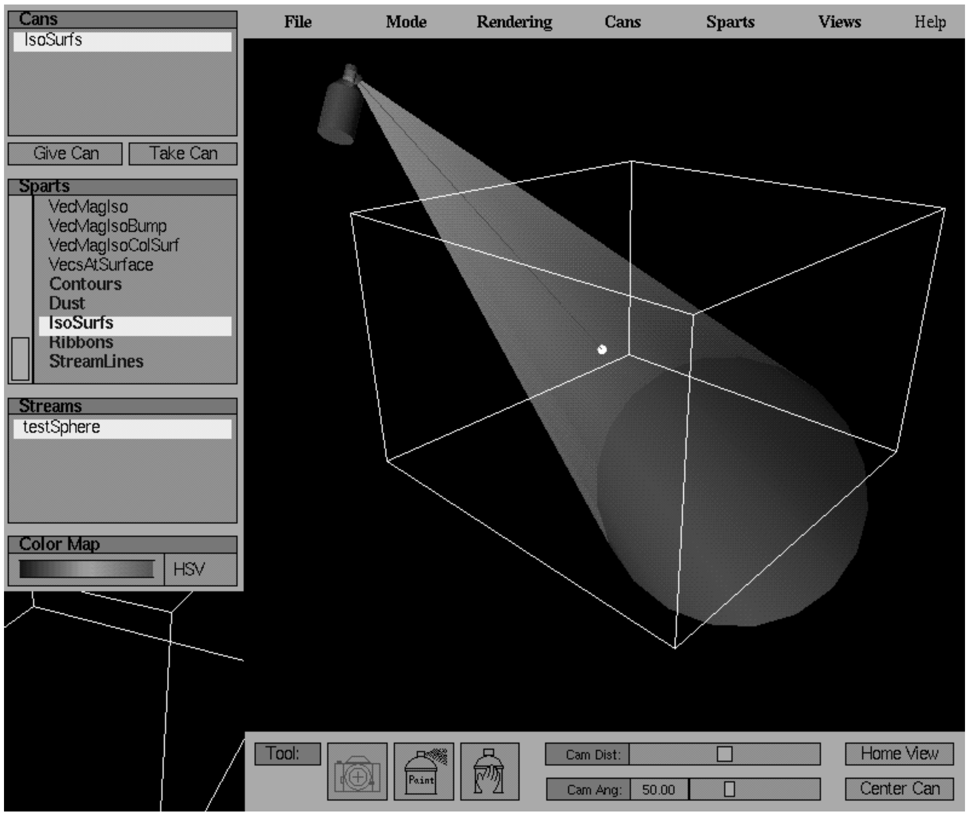


Figure 4.3: This image shows the top-level user interface of *Mix&Match*. The main window shows the current state of the visualization. Here, the bounding box of a stream is shown as well as a spray can with a conical nozzle. The smaller window in the lower left corner is the view from the view-point of the can.

four categories that reflect the stages in the life-cycle of a typical spart. *Target* functions look for target features and output booleans, *Visual* functions deposit AVOs if certain conditions are satisfied, *Position* functions update the current position of the spart and *Death* functions determine when the spart should die. Each of the components making up a composed spart is a function that gets executed at the current location by the spart executive until the spart dies.

Sparts are visualization methods or tools. The ability to define new sparts from basic components allows experimentation with different techniques. However, since this flexibility comes at the expense of efficiency, the system also allows for more efficient but non-

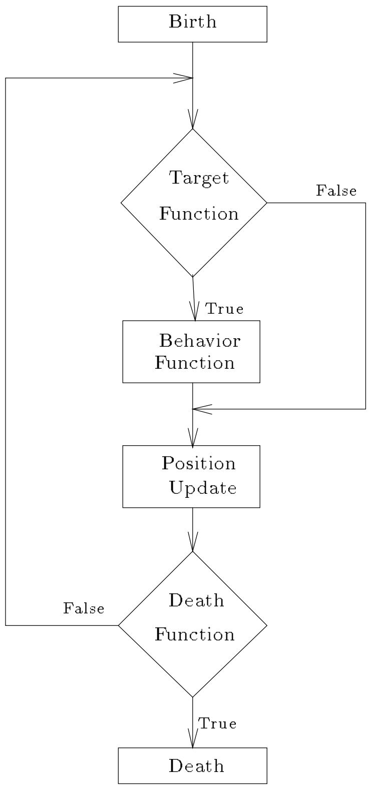


Figure 4.4: Flow diagram illustrating the life-time of a typical spart.

modifiable sparts called *Predefined* sparts. These are monolithic sparts that embody all stages of the life-cycle in a single function. Once launched, they return control to the spart executive only when they die.

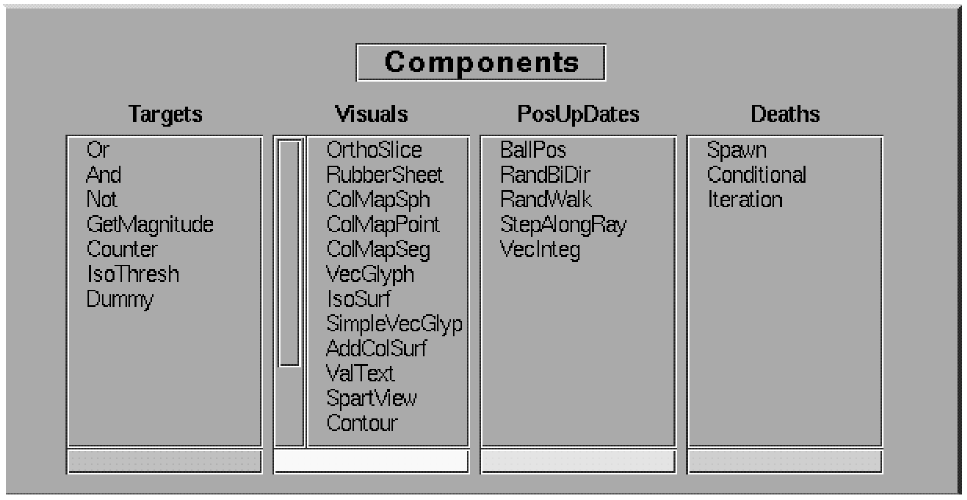
4.4 Spart Composition

The user can compose sparts either using a textual editor or a graphical editor. A composition is the specification of the components that make up the spart and the connections between them. Since this is basically the specification of a program, the two options reflect the textual and visual programming paradigms.

4.4.1 The Textual Interface

The primary goal in the design of the textual interface was ease of use. Rather than presenting a full text editor and a complicated language for program specification, the simplicity of the visual programming style of data-flow visualization environments was employed. Since the "language" for program specification is quite simple, the interface retains the "pick and drop" nature of the graphical interface. Users can, of course, use their favorite editor off-line to compose a spart since the spart definition format is ASCII and quite simple.

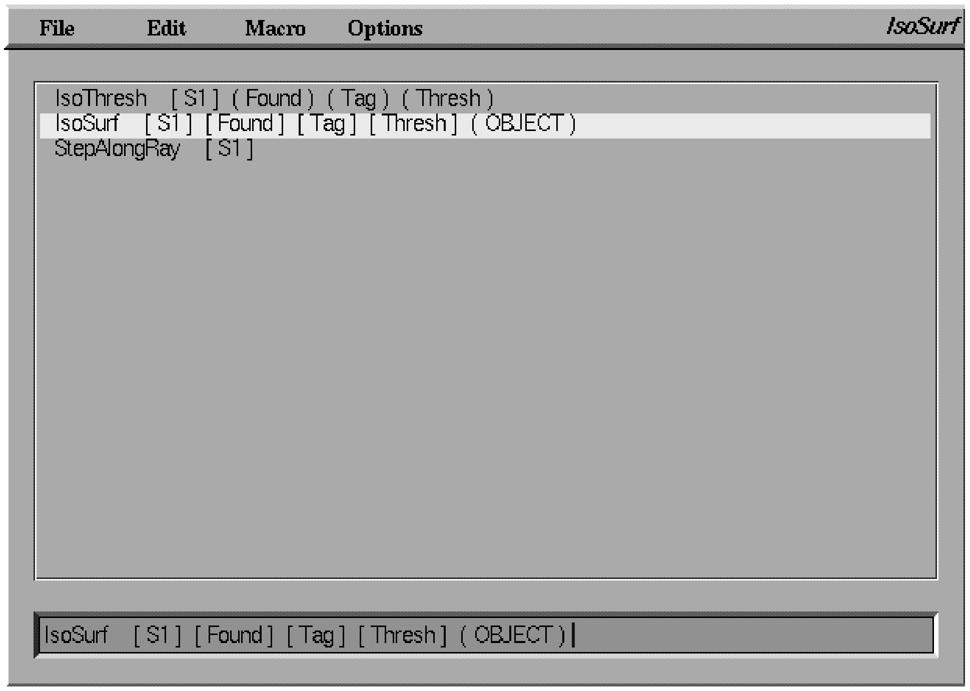
In the textual mode, the user is presented with the collection of components arranged in four browsers (figure 4.5), one for each category. There is also a main editing browser where the composition takes place that consists of an input field where a line can be edited and a main browser that displays the composition. Figure 4.6 shows the main browser with the completed composition of the iso-surface spart.



Targets	Visuals	PosUpDates	Deaths
Or	OrthoSlice	BallPos	Spawn
And	RubberSheet	RandBiDir	Conditional
Not	ColMapSph	RandWalk	Iteration
GetMagnitude	ColMapPoint	StepAlongRay	
Counter	ColMapSeg	VecInteg	
IsoThresh	VecGlyph		
Dummy	IsoSurf		
	SimpleVecGlyp		
	AddColSurf		
	ValText		
	SpartView		
	Contour		

Figure 4.5: The components listed in four browsers, one for each category.

When a user selects a component from the components browser, the name of the component selected appears in the main browser together with its number of inputs and



```
IsoThresh   [ S1 ]  ( Found )  ( Tag )  ( Thresh )
IsoSurf   [ S1 ]  [ Found ]  [ Tag ]  [ Thresh ]  ( OBJECT )
StepAlongRay   [ S1 ]
```

```
IsoSurf   [ S1 ]  [ Found ]  [ Tag ]  [ Thresh ]  ( OBJECT )
```

Figure 4.6: The text based spart editor showing the iso-spart composition.

outputs and their types. For example, if the user chose the `IsoThresh` component from the targets category, the line that would appear in the browser would be

```
IsoThresh [ STREAM ] ( BOOLEAN ) ( INT ) ( FLOAT )
```

The inputs are enclosed in square brackets and the outputs in regular parenthesis. The types appear in this enclosure in capital letters. The currently selected line from the main browser also appears in the input field below where it can be edited. The user can thus select the components that will appear in the composition and then specify the connections by editing the lines one by one. The connections are specified by naming the input and output fields of the component. These names are actually names of connections. For instance, the connections between the `IsoThresh` and the `IsoSurf` components can be specified by the following:

```
IsoThresh [ S1 ] ( SurfFound ) ( Tag ) ( IsoVal )
IsoSurf [ S1 ] [ SurfFound ] [ Tag ] [ IsoVal ] ( OBJECT )
```

The output fields of the `IsoThresh` component have been given the same names as some of the input components of the `IsoSurf` component. Thus the name `SurfFound` specifies a connection between the first output field of `IsoThresh` and the second input field of the `IsoSurf` component. Input fields can also be specified to be constant valued. This is identified by having the '=' sign precede the value. For instance a boolean field may be tied to a constant true value by [=TRUE].

When a composition is completed, it can be parsed for correctness. The main rules for a correct composition are as follows:

- All the inputs must have either a constant value specified for them, or they need to be tied to the output of a component. In other words, there are no optional inputs. Output fields may be left unspecified, however.
- The output and input fields that are connected must have the same type. There is thus strong typing but no type coercion.
- Since the parser is one-pass, the components that provide input to other components must precede them. In other words, if a directed graph were to be constructed from the components where the edges denoted dependency, the components in the main browser must be in the topological order of this graph.
- Fan-out is allowed, while fan-in is not. In other words, there cannot be input fields that take input from more than one output field.
- The dependency graph between components must be acyclic.

An example of a spart composition is the `IsoSurface` spart below which extracts iso-valued surfaces from volumes.

```
IsoThresh [ S1 ] ( SurfFound ) ( Tag ) ( IsoVal )
IsoSurf [ S1 ] [ SurfFound ] [ Tag ] [ IsoVal ]  ( OBJECT )
StepAlongRay [ S1 ]
```

The `IsoThresh` is the target behavior function, `IsoSurf` is the visual behavior function and `StepAlongRay` is the position update function. There is a default death function that kills the spart when it exits the bounding box of the stream.

4.4.2 The Graphical Interface

The graphical interface provides a more intuitive way of specifying the spart composition than is provided by the textual interface. Typing is limited to specifying constant input values. Everything else is mouse driven.

The user has the same components browser available as with the textual interface. Instead of the text based editor, there is a canvas for graphically editing the composition (figure 4.7). A user can select a component from the components browser and position and drop it onto the canvas. The module appears as a colored box in the canvas with two parts. The top half displays the name of the module and an expansion button, while the bottom half displays two boxes that display menus when selected by the mouse. The graphical editor possesses all the editing capabilities of object based drawing programs such as multiple selection, moving, copying, cutting and pasting.

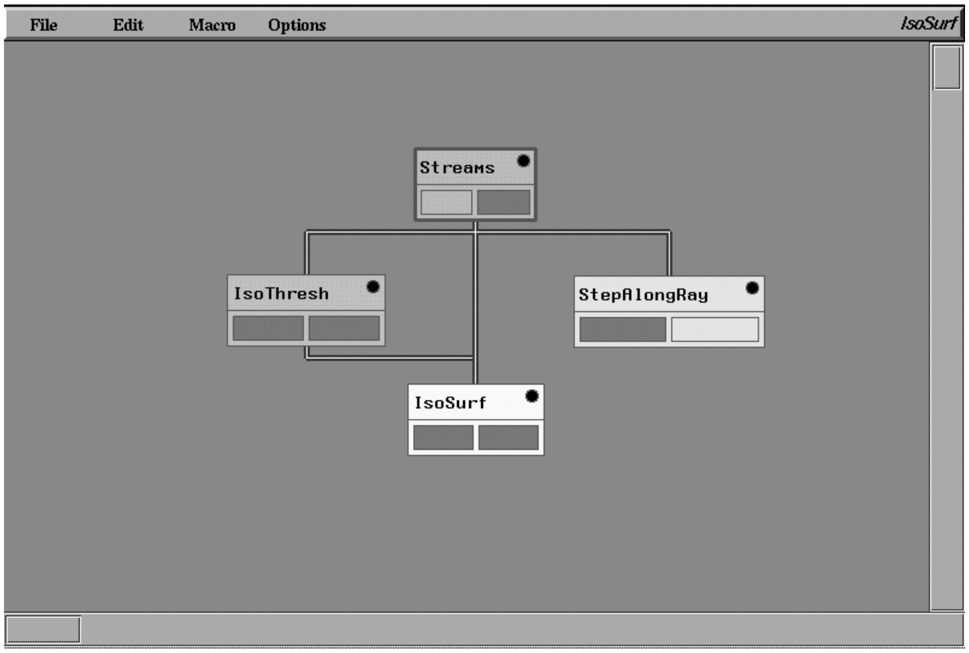


Figure 4.7: The graphical spart editor showing the iso-surface spart composition.

The process of specifying the connection between two components is to select the outputs menu from one component and choose the particular output field from it. At this stage all the components on the canvas which have an untied input that is type compatible with the selected output field will have their input menu highlighted. If the automatic expansion flag is set, these components are expanded to show the compatible input fields so as to assist the user in the selection of a matching input field (figure 4.8). If an input field is chosen that is not type compatible, an appropriate error message is displayed immediately. Thus, the user gets an earlier warning than the textual editor in this regard.

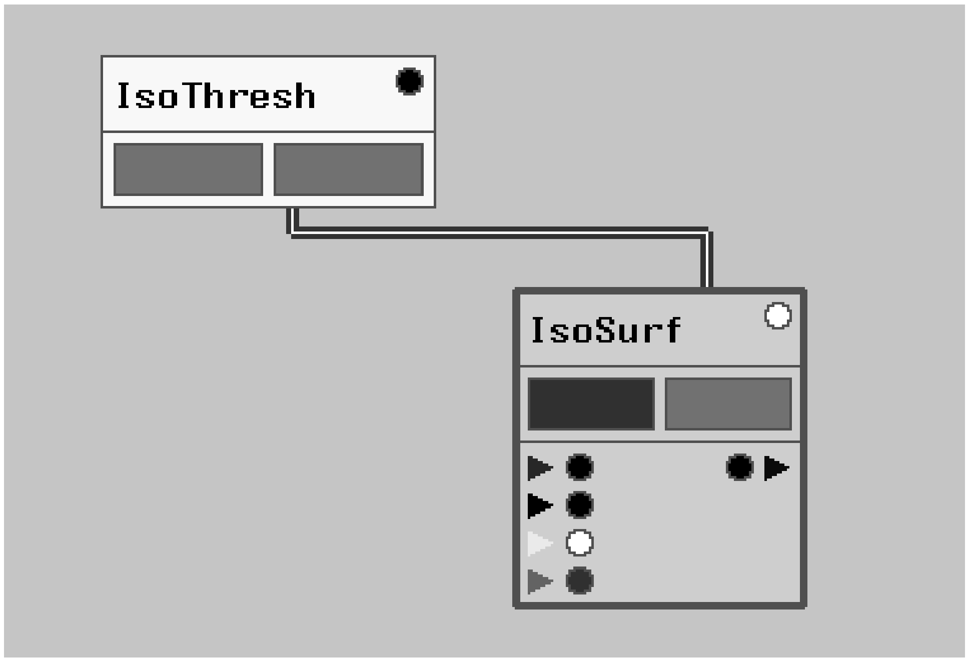


Figure 4.8: Detail showing how a box has expanded to reveal its compatibly typed input highlighted (green circle) for a potential connection. The left menu box has also been highlighted. The expansion also shows which fields have connections (black circles) and which ones don't (white circles). The color coded triangles represent the types of the fields.

If the connection is valid, then submenu items at both ends of the connection will be added to the input/output menus and a "wire" will connect the boxes. The submenu item will indicate the other end of the connection. For instance, the same connections between the `IsoThresh` and `IsoSurf` components as was presented in the previous section will result in the following menu items:

```
IsoThresh
  Inputs menu:
     Stream:      Tie
                  <- Streams(1).S1

  Outputs menu:
     SurfFound:   Tie
                  -> IsoSurf(1).Execute
     Tag:         Tie
                  -> IsoSurf(1).Tag
     Threshold:   Tie
                  -> IsoSurf(1).Threshold

IsoSurf
  Inputs menu:
     Stream:      Tie
                  <- Streams(1).S1
     Execute:     Tie
                  <- IsoThresh(1).SurfFound
     Tag:         Tie
                  <- IsoThresh(1).Tag
     Threshold:   Tie
                  <- IsoThresh(1).Threshold

  Outputs menu:
     Surface
```

The first item in each submenu is the `Tie` which, when selected, adds a new item to the submenu. An already established connection can be disconnected by re-choosing the connection item in the submenu. Note that since there can only be a single input to a module, the `Tie` entry of an input submenu that has a connection is unselectable. The `Stream` inputs of the components are tied to a stand-in component called `Streams` that has streams as output. This way, the stream inputs of components are given variable names that can later be bound to loaded streams.

4.4.3 Component Hierarchy

It is possible that a user will often use a particular set of components in combination and wishes to use this set as a single component in spart compositions. The system allows the user to create such components called *macros* both in the textual and the graphical composition interfaces.

In the textual interface, when the user defines a macro, the set of components is replaced by the macro in the main browser, and the macro is added to the components browser. Which particular category the macro is added to is specified by the user. The inputs and outputs of the macro are those that were not tied. A separate macro browser allows the displaying, editing and saving of macros.

In the graphical interface, the user is able to make a subset of the components on the canvas into a macro. The components are then replaced and the connections to the remaining components preserved. A separate macro canvas allows the displaying, editing and saving of macros.

Internally, the macros are expanded into their constituent components at the time the spart is loaded into the can. Hence, the restriction on component dependency between components of different categories still applies. A macro behaves just like a component and can be used in compositions. Macros can also include other macros as long as the inclusions do not result in a cycle.

4.5 Interaction Modes

In *Mix&Match*, visualization is an interactive and incremental process. The user selects spray cans and delivers doses of sparts into the data set, leaving visual objects behind. The sparts conform to a basic mode of operation: they seek targets, deposit visual objects, and update their position. This is repeated until the spart dies. This general behavior allows the control of some parameters that results in different and interesting modes of interaction. These interactions emphasize the exploratory nature of the visualization process. Table 4.1 summarizes the options that can be set to achieve these interactions.

There are three main categories for the options:

1. **Spart Delivery.** This category determines how the sparts are to be delivered or launched. In spray rendering the instrument of delivery is the spray can. For discrete

Spart Delivery	Rate of Delivery	AVO Persistence
Spray	On Mouse Down	Accumulate
Grid	While Mouse Down	Keep #ofAVOs
	Continuous	Update #of Sparts
	While View Change	Update #of Deliveries
	While Parameter Change	Update #of Positions
		While View Change
		While Parameter Change

Table 4.1: This table summarizes the options available for achieving different interactions. A choice is made from each of three orthogonal categories. *Spart Delivery* determines whether the sparts are to be delivered from the can or they should traverse the grid. The *Rate of Delivery* determines how often new doses are delivered while the *AVO Persistence* category determines how long visual objects should remain in the scene.

visualization techniques, this is suitable and an intuitive metaphor. For continuous techniques, grid traversal offers a more robust delivery method. For instance, with iso-surface sparts, if surfaces were generated discretely, i.e., wherever the sparts passed and satisfied the iso-value criterion, the absence of a surface could not definitely be attributed to the criterion not being satisfied. It may be that the spart missed the region and could not generate the surface in that region. Grid traversal ensures the visiting of all cells in a specified region. There are, then, two options for spart delivery:

- *Spray.* In this case, the sparts are launched from the can. For this reason, the user can choose a nozzle shape and size and also specify the number of sparts per delivery. The basic nozzle shapes available are *point, line, circle*, and *square*. By default, these shapes correspond to a line, a triangular plane, a circular cone (see figure 4.3) and a square pyramid in 3D respectively. If another option is set, the 3D nozzle shapes become a line, a rectangular plane, a cylinder and a parallelepiped. Sparts are constrained to be delivered from within the nozzle.
- *Grid.* In this mode, only a single spart is delivered and it traverses the grid. Any position update components and death functions in the composition are ignored. There are options that can be set in grid mode: we can traverse *All* of the grid, a *SubVolume* or a *BallRegion.* When a subvolume is to be chosen,

the user specifies the bounding region in computational space. Similarly, a single number defines the extent of the region around the ball tethered to the can[2]. In all cases, the user can specify *subsampling* factors for each axis. This means that only every nth sample will be taken during traversal along that dimension. Users can also choose one axis along which to *animate*. Grid traversal occurs with the i dimension varying fastest and the k the slowest. This means that, if the animate button for the i dimension is chosen, the view will be updated at every position visited during traversal. If the j or the k buttons are chosen, the view will be updated every time those indexes change.

2. **Rate of Delivery.** These options define how often the sparts should be delivered. The options are as follows:

   - *On Mouse Down.* In this mode, one dose will be delivered when the mouse is first pressed. No more doses are delivered until the mouse is released. This mode is useful when the spart delivery mode is grid. It ensures that only one spart will be delivered when the mouse is pressed.
   - *While Mouse Down.* New doses are delivered every time (through the main event loop) as long as the mouse is pressed. This is the "drag and keep" spraying mode.
   - *Continuous.* In this mode, the user does not need to keep pressing the mouse. New sparts will be delivered every time through the main event loop. This is useful when in an animation mode.
   - *While View Change.* Another way to launch sparts is to do so when the user is changing the view. Interesting results can be obtained if the can is optionally fixed at the center of the view. This may prove useful for a direct volume rendering spart. Another use for this mode is that the view can be changed as a position-based animation is taking place.
   - *While Parameter Change.* Most components making up a spart have parameter widgets associated with them. It is very informative to launch sparts as the

[2] The spray can is drawn graphically in 3D and can be interactively positioned. It has a sphere at the end of its directional axis for an alternative means of manipulation (see figure 4.3).

parameter is being changed and the visual objects are updated at the same time. For instance, if the redrawing is fast enough, the scale factor parameter of a vector glyph widget can interactively be changed to find just the right relative sizes for a particular view. As another example, the iso-surfaces at different iso-values can be generated as the iso-value widget is varied interactively.

3. **AVO Persistence.** Another category of options determines how long the visualization objects are to be kept in the scene. The options are:

   - *Accumulate.* Any new objects produced are saved and accumulate in the scene.
   - *Keep #of AVOs.* Alternatively, only a specified number of objects can be kept. This operates as a FIFO, and as new objects are added, others are taken off the doubly linked list. The result is a flashlight-like effect. Note that this operation does not affect what has already been accumulated. Only a working buffer is affected. If, later, the user switches to the accumulate option, the working buffer is added to the accumulated objects.
   - *Update #of Sparts.* In this mode the working buffer is deleted every nth spart delivered. There is no spart FIFO.
   - *Update #of Deliveries.* In this mode the working buffer is deleted every nth delivery. For instance, delivering a single spart and updating every delivery results in a probe like interaction.
   - *Update #of Positions.* This is a unique mode. Not only is the working buffer updated every nth position (a position being the spart location during the iterations of its life-cycle) but the scene is also redrawn. This mode is therefore used for simple position-based animations. For instance, an ij slice can be produced at every $(0, 0, k)$ node. When in this mode, the slices will be animated from $(0, 0, 0)$ to $(0, 0, k_{max})$. The speed of the animations can be controlled by specifying a *sleep* parameter.
   - *While View Change.* This can be used with its corresponding rate of delivery option to update the working buffer as the view is changed.

- *While Parameter Change.* Similarly, this can be used with its corresponding rate of delivery option to update the working buffer as a parameter widget is changed.

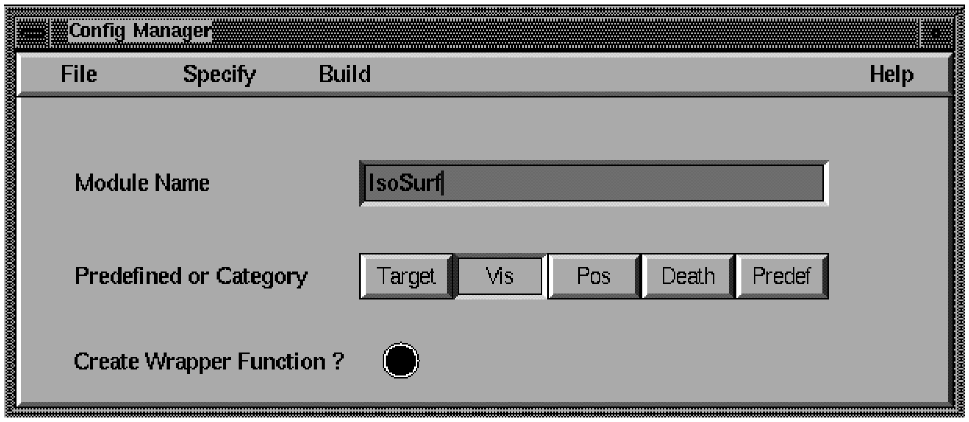
4.6 Extending the System

If there are many components that can be composed to cover the common visualization techniques, intermediate users will merely be concerned with the compositional aspects. Novice users do not even have to deal with that complexity, since they can just load compositions or predefined sparts into the cans. Expert users, on the other hand, will want to extend the system by writing their own components. Extensibility is very important since there will almost certainly be a need for a component that is not provided initially.

Component writers write some functions (in C) as a separate module and compile and link it with the application. A number of application programmer interfaces (APIs) are provided for accessing and manipulating system internals. The task of describing the new component to the system is made easier by a separate program called the Configuration Manager (CM). It presents an easy-to-use graphical interface to the process of defining what the particulars of a component are, and then generates code for this component. Using the CM, definitions can be recalled and modified graphically rather than by manually editing files.

When CM is first run the user is presented with the top-level window in figure 4.9. The user specifies the name of the component, what category the component will belong to, and whether wrapper code for the top-level user function is required. Next, the user can proceed to specify the inputs, outputs, state variables, the user functions and the parameter widgets the component needs.

For the inputs, outputs and state variables, the user enters the names of these variables and their types. The names are used during graphical spart composition. Also, there exist convenience functions that return the index into the corresponding arrays based on the names. The component writer is thus able to reference these with names rather than indexes which may change as the component is modified (figure 4.10).

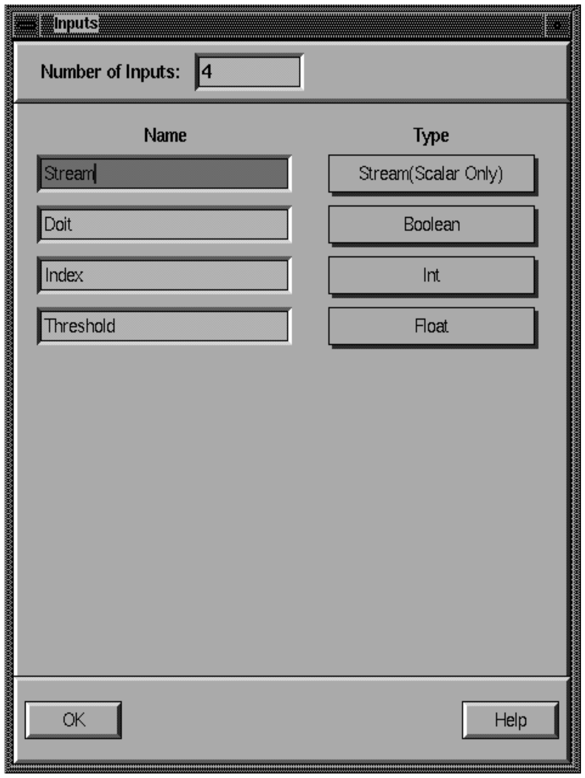


Module Name: IsoSurf

Predefined or Category: Target [✓ Vis] Pos Death Predef

Create Wrapper Function ? ●

Figure 4.9: The top-level window of the configuration manager. The round button indicates that wrapper code is desired.



Number of Inputs: 4

Name	Type
Stream	Stream(Scalar Only)
DoIt	Boolean
Index	Int
Threshold	Float

Figure 4.10: The window to enter and indicate the number, names and types of the input variables of the component.

The user specifies the function names such as the top-level user function and the initialization function. The top-level user function is required while the others are optional. Finally, the user interactively designs the control panel where the parameter widgets are to reside. The task of creating a panel and placing the control widgets can be time consuming and error prone if done through procedure calls, especially when one considers that the panels may evolve over time and go through changes. For this reason, the design of the control panel is graphical and interactive.

When the user is satisfied with the specifications, the component definition can be saved. There exists one ASCII definition file for each component. When the program is compiled to incorporate the module into the system, these component definition files are gathered into an array so that component particulars can be accessed at runtime.

5. Implementation

The previous chapter presented a high level view of the *Mix&Match* environment. In this chapter, some of the more important implementation details are discussed.

5.1 Data Streams

All scientific visualization environments operate on native data types based on a data model of scientific data. In this section, the structure of the main data type of *Mix&Match* called a *stream* is described.

Figure 5.1 shows the data structure of a stream. There are two types of streams, as

```
typedef struct _streamList {
    Str                         title;          /* name of stream */
    StreamType                  strType;        /* type of stream */
    union {
       Structured               lat;
       Unstructured             fem;
       } t;
    CoordSysPtr                 coordSys;       /* coord system */
    Bool                        needToConvert;  /* need conversion ? */
    int                         inUse;          /* stream in use by a spart */
    struct _streamList          *next;
    } StreamList;
```

Figure 5.1: Data structure of a stream.

suggested by the union in this structure. *Structured* streams are basically multidimensional arrays of single scalar or vector values. The *Unstructured* streams would contain explicit connectivity information, but are not supported by the stream API routines currently implemented.

Since *Mix&Match* was designed primarily for meteorological visualization there is the concept of a coordinate system (a map projection) for the data (`coordSys` member). Meteorological simulation data assumes that a specific projection has been used during the simulation. This projection becomes the native coordinate system of the data. The user can visualize the data either in its native projection or in another projection in which case transformations take place on the fly as the spart is traveling in the current coordinate system.

The structured stream's data structure is shown in figure 5.2. Redundant information is kept in the structure so that frequently required quantities need not be computed by the spart components. This is important because the components are fine grained and will be called many times during the life-time of a spart (e.g. the `xysize` is derivable from the `dims` array). Index arrays assist in the grid traversal mode of spart execution. The coordinate type of structured streams define whether they are *regular*, *rectilinear* or *curvilinear*.

```
typedef struct _lattice {
    int                         nDim;           /* number of dimensions */
    int                         *dims;          /* dimensions in each dir */
    long                        xysize;         /* the prod of x and y dims */
    int                         curIndexes[3];  /* current comp space indexes */
    int                         lows[3];        /* Min indexes of sub-volume */
    int                         highs[3];       /* Max indexes of sub-volume */
    int                         subs[3];        /* subsamples */
    Data                        data;           /* the data array */
    Coords                      coords;         /* the coordinates array */
    } Structured;
```

Figure 5.2: Data structure of a structured stream.

The spart components only operate on single scalar values or on a vector of two or three values. Having only these types to operate on keeps the component code simpler. In many cases, users are only interested in a few of the parameters of a multiparameter data set at any one time. This way, the whole data set does not need to be loaded. For instance, the NORAPS model that has been used during this study generates 12 two dimensional and 6 three dimensional parameters for a single simulation time. Loading all of the parameters is not usually necessary and would be wasteful of precious RAM. Further, separation of parameters allows different position update functions and death functions based on different parameters of the same data set.

Another tradeoff between memory and execution time is the saving of scaled data. Many visualization behaviors use a color map to map the data values to color. The data values are scaled to the range 0-255 and the value used as an index into a 256 bucket color table. To save execution time, data are prescaled and stored as part of the stream.

5.2 The Spray Can

One of the main ingredients of spray rendering is the virtual spray can which is used to focus and deliver the sparts into the data set. It forms the tool that is used for interaction. Depending on what attribute and mode settings have been specified, different interactions can be achieved. The spart contents of the can determine the visualization result of these interactions and essentially define the nature of what the tool is. In this section, the important aspects of the can structure are described in detail. The complete structure is shown in figure 5.3.

```
typedef struct _can {
   FourVector      pos;                /* position */
   FourVector      dir;                /* direction */
   float           CanCenterPointDist; /* distance to the ball */
   float           NozzleTwist;        /* nozzle twist */
   int             SpartDist;          /* spart distribution */
   Bool            OrthoSpray;         /* spart directions parallel ? */
   ObjList         AVOS;               /* head of the AVOs */
   ObjPointers     *UndoStack;         /* the for undo operations */
   GeoObject       *AVOSLastKept;      /* pointer to the accummulated avos */
   Bool            DrawAVOs;           /* draw AVOs ? */
   Bool            DrawCan;            /* draw can ? */
   Color           *colMap;            /* color map for this can */
   int             colMapName;         /* name of color map */
   SprayMode       sprayMode;          /* spray mode */
   Str             spartName;          /* spart name loaded */
   ProgramList     *programs;          /* the programs */
   Bool            lock[4];            /* position locks on can */
   strcut _can    *next;
   } Can;
```

Figure 5.3: Structure of a can.

When a can is created, the currently selected spart from the spart browser becomes its content and is identified by `spartName`. The can has a position in world space and can be moved and oriented. It is placed at the center of the current view when first created. The `dir` element forms the axis of the cone associated with the can, and indicates the general direction in which the sparts will be delivered. The `CanCenterPointDist` element specifies the distance to a ball that is tethered to the can for an alternative means of orienting the can. There are several types of nozzles that can be chosen and their sizes can be set. The user can thus aim and focus the can into a particular area of interest and release sparts selectively. The number and the pattern of distribution of sparts can also be set. The display of the can and its cone may be switched on and off.

The sparts will manifest themselves in the form of abstract visualization objects (AVOs) and get added to the can's `AVOS`. This is a doubly linked list of objects that can be optionally rendered. The `DrawAVOS` flag is used to temporarily hide the AVOs so as to prevent image clutter, or to turn it on and off to compare different spart manifestations. Considering that the user may have created several cans which may have hidden or visible AVOs, at a certain stage of the visualization the renderer renders the visible cans and the visible AVOs. The scene that is rendered is summarized in figure 5.4 where the solid boxes are visible cans or AVOs and are rendered, and the dashed boxes stand for cans and AVOS that have been temporarily hidden.

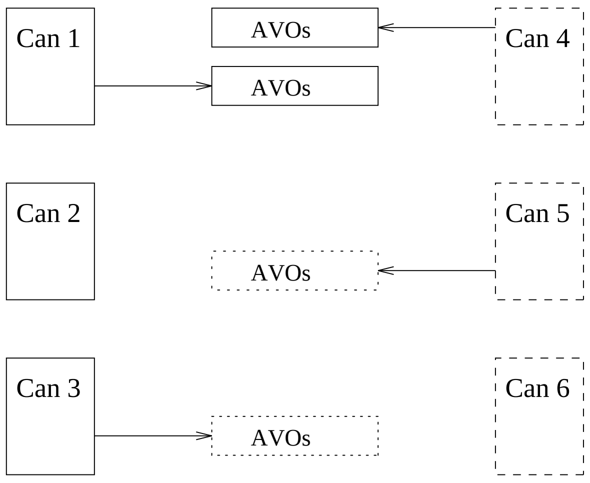


Figure 5.4: This figure illustrates the scene that is rendered. Solid boxes are visible cans and AVOS and dashed boxes are those that have been temporarily hidden.

Another way of uncluttering an image is to undo some of the spraying. Each dose of spray is pushed on to the `UndoStack`. Users can undo a dose of spray by popping this stack. The `AVOSLastKept` member is used as the pointer to the accumulated AVOs. Any probe-like behavior (as discussed in section 4.5) does not affect the accumulated AVOs.

A non-editable color map from a list of color maps is associated with each can. The spart components that use one for mapping data values to color use the can's color map. This limits all the components making up the spart to a single color map. Alternative schemes could be to:

- associate one color map with each stream. This would require the binding of a color map to a stream when the latter gets loaded. More than one color map would then have to be displayed as feedback if a particular spart is using multiple streams with different color maps.
- make the color map an editable widget associated with each component that needs one. The setting of each component's color maps would be time consuming and the component code would be more complex.
- define a color map type and write a component that produces an editable color map. The output of this component can then be sent as input to components that require it. The composition would be more complex and the spart execution slower.

The `sprayMode` element is used to obtain different interactions with the can. Controls establish choices for whether the sparts should be sprayed from the can or traverse the grid, the type of delivery rate and also the persistence of the visualization objects. Combinations of these choices result in very different interaction techniques using the same tool as discussed in section 4.5.

Perhaps the most important element in this structure is the `programs` element. This is what characterizes the contents of the can, namely the particular spart it contains. The spart may be a predefined spart or a composed spart consisting of components. As the name suggests, the spart's overall behavior from birth to death is in effect a program that is executed when the spart is launched from the can. Sparts are discussed in greater detail in the next section.

5.3 The Sparts

Aside from the spray can, the other main ingredient of spray rendering is its contents, the smart particles or *sparts*. These determine the visualization. How the spart behaves is determined by a program that is associated with the can, so that different cans loaded with different sparts will produce different effects. The sparts, then, define the kind of tool each spray can is.

There are two kinds of sparts. *Predefined* sparts are ones that are unmodifiable at runtime. They are monolithic and self contained in the sense that once control is given to the spart (it is born), it will return to the spart executive only when it dies. *Composed* sparts, on the other hand, are made up of components which do not know anything about each other. The program consists of the repeated execution of these components in a certain order until the spart dies. Before going into more detail about the two types of sparts, an analysis of the `ProgramList` data structure that appears as an element in the can structure will be given.

The `ProgramList` data structure is shown in figure 5.5 The dichotomy between a predefined and a composed spart is immediately obvious from the union that appears in this structure. The reason for having a list of programs in the can's data structure rather than a single program is that sparts may spawn new sparts of a different kind during their lifetime. In the case of a composed spart, this may be achieved by having the *Spawn* component appear in the composition (see section 6.1.4).

The `spartPool` element is the queue of sparts that are to execute the particular program. The `SpartList` structure is simply a list of position and direction pairs that indicate the initial state of the spart at the time of its release. In a predefined spart, the top-level function uses this information and decides what to do with it. In a composed spart, the state of the current spart is updated by a position update function.

```
typedef struct _programList {
  Str                   name;           /* name of spart */
  SpartList             *spartPool;     /* queue of sparts to execute */
  Bool                  predef;         /* is it predefined ? */
  FormsWindow           formsWindow;    /* widget container */
  union {
     Predefined         predefSpart;    /* predefined spart */
     Composed           compSpart;      /* composed spart */
     } p;
  struct _programList   *next;
  } ProgramList;
```

Figure 5.5: Structure of `ProgramList`.

5.3.1 Predefined Sparts

Predefined sparts are self-contained, monolithic sparts. When one is launched from the can by the spart executive, what it will do and when it will die are all determined internally. All the different categories of components that appear in a composed spart are embodied in it.

The structure of a predefined spart is shown in figure 5.6. A predefined spart is registered with the system by providing the values for the members of this structure.

```
typedef struct _predefSpart {
  Str            name;                          /* name of spart */
  int            noStreams;                     /* number of streams */
  Str            inNames[MAX_MOD_INPUTS];       /* input names */
  InOut          inTypes[MAX_MOD_INPUTS];       /* input types */
  StreamList     **streams;                     /* pointers to the streams */
  int            noInterVars;                   /* number of internal vars */
  Str            interVarNames[MAX_MOD_INOUTS]; /* inter var names */
  InOuts         interVarTypes[MAX_MOD_INOUTS]; /* inter var types */
  void           **interVars;                   /* pointer to inter vars */
  void           (* behav)(PredefinedPtr predef);/* the behavior function */
  void           (* init)(PredefinedPtr predef);/* initialization function */
  void           (* createForm)(PredefinedPtr predef);/* creates the form */
  void           (* setForm)(PredefinedPtr predef);/* sets the form */
  void           (* getParameters)(void);       /* gets the parameters */
  Widget         *paramWdgt;                    /* parameter widgets */
  } Predefined;
```

Figure 5.6: Structure of a predefined spart.

The `noStreams` element specifies the number of streams that the predefined spart operates on. These will be bound by the user to currently loaded streams at the time the spart is loaded into the can. The names and types of streams are provided so that streams can be identified and accessed through convenience functions. The types ensure type compatibility when streams are bound.

Internal state variables are provided so that the execution of the predefined spart can depend on its state. This is necessary since there may be multiple instances of a predefined spart.

The most important element of the structure is the `behav` element, which is the top-level call to the spart. This function will be called once, and it will return when the spart dies (having generated its AVOs). The function `init` does internal state initialization. The `createForm` function creates the control panel holding the parameter widgets that the spart may require. The function `setForm` sets the initial state and values of the widgets contained in the control panel and the parameters are obtained by the `getParameters` function.

It is helpful to look at the execution model for a predefined spart at this stage. The sparts are delivered in doses determined by the spray mode settings (see section 4.5). These settings determine the number of sparts and their initial positions and directions. The sparts to be delivered are added to the can's program's spart pool. The spart executive processes this pool as described in the pseudocode in figure 5.7.

There are several things of note in this pseudocode. The widget parameters are obtained once for all the sparts in the dose of delivery. This is more efficient than having the top-level functions obtain their parameters at each call when there are many sparts in a single delivery.

The "while" loop is necessary because the can's program list may grow due to spawning. If the predefined spart is to spawn sparts that are of the same type, these are merely added to the spart pool of the current program. Otherwise, a program is added to the can together with its spart pool.

The initial position and direction of the spart is used to intersect it with the bounding boxes of the streams it depends on. These entry and exit point pairs need to be processed by

```
void sendPredefSparts(void) {
     /* get the widget parameters */
     for (each program in the can's program list)
        call getParameters function;

     /* handle the programs */
     while (not done) {
        get a program from the can that has a non-empty spart pool;
        if (no such program exists)
           we are done;
        else {
           for (each spart in the spart pool) {
              if (AVO persistence mode is number of sparts type)
                 delete the AVOs in the working buffer;
              if (spart delivery mode is grid type) {
                 call initialization function;
                 call behav function;
                 }
              else {
                 get the initial position of the spart in all its streams
                 if (spart alive) {
                   call initialization function;
                   call behav function;
                   }
                 }
              remove the spart from the spart pool;
              }
           }
        }
     }
```

Figure 5.7: Pseudocode for the execution of predefined sparts.

the spart if it depends on more than one stream. If the ray does not intersect the bounding box of any of the streams the spart is dead and can be removed from the spart pool.

The sparts are handled sequentially until their death. This assumes that their behavior does not depend on the states of other sparts, i.e. there is no communication between them.

5.3.2 Composed Sparts

Composed sparts are constructed from independent basic building blocks. The user composes a spart by connecting the inputs and outputs of these components. The overall behavior of the spart is determined by the combination of these components. The components are executed at each location the spart occupies during its life-time, exchanging data in the process. Since they operate locally, their granularity are fine.

These components have been based on ideas from spray rendering. A typical spart has a life-time as depicted in figure 4.4. Once a spart is born, it may search for target features in the data set. It may then produce AVOs that can be rendered. These behaviors take place at the current location of the spart. The spart may then update its position and decide whether to die or not. Note that the figure has been simplified. For instance, there may be sparts that do not have a target function and whose visual behavior function executes unconditionally, or there may be death functions that depend on multiple conditions. In the light of this flow diagram, the components have been organized into four categories.

1. *Target* behavior functions are feature detection components. They usually test to see if a condition is satisfied at the current location of the spart. A boolean output is set accordingly.
2. *Visual* behavior functions are the key visualization components. They are responsible for the output of the AVOs. Because they should take effect conditionally, each has a boolean input. They can then be executed only if a target function is satisfied. They usually make the AVOs they produce available as output so that other components can operate on them.
3. *Position* functions update the current position of the spart. These can be absolute or dependent on the data as in vector fields. They can also be nondeterministic.
4. *Death* functions determine when the spart should die. There is also a birth function in this category that spawns new sparts.

```
typedef struct _program {
  ModList      *mods[NO_MOD_CATEGORIES];  /* modules in the composition */
  Map          *map;                      /* address mappings */
  int          noStreams;                 /* number of streams it needs */
  Str          names;                     /* their names */
  StreamList   **streams;                 /* the streams */
  } Composed;
```

Figure 5.8: Structure of a composed spart.

The composed spart program structure (figure 5.8) is built when a composed spart is loaded into the can. The array `mods` holds pointers to the list of components (or modules) in the composition each in its own category. The `ModList` is simply a list of `Module` structures. The `map` element is used to do the mapping between the inputs and outputs of the modules. The streams that the composed spart will operate on also appear in this structure.

As mentioned in section 5.3.1, it is the spart executive that processes the spart pool and executes the program. In the case of a composed spart, the pseudocode is as in figure 5.9.

There are several things that distinguishes this pseudocode from that of predefined sparts. Since a composed spart is made up of components, each of these components execute. The target components are executed first followed by the visual, position and death components. Within each of these categories, the modules execute in the order of their data dependencies.

While the predefined sparts handle the grid traversal mode internally, the components must rely on the spart executive. In this mode, the position update and death functions of the composition are ignored. Instead, the spart executive updates the current position of the spart based on the walk through of the computational grid. When the grid region of interest has been traversed, the spart dies.

In grid traversal mode, the user can choose to turn on animations based on the grid traversal. The scene can be updated at every change of value of the i, j, k indices of computational space. For this to take effect, the AVO persistence mode has to be such that the AVOs are deleted depending on the number of positions. In spray mode, the animation again takes place if the AVO persistence is in this mode. In other words, every so often along a spart's path, the scene is updated and the AVOs generated so far are cleared.

The structure of a component is called a `Module` and is very similar to the predefined spart structure (figure 5.10). The differences reflect the fact that modules are used in compositions. They therefore have outputs as well as inputs and internal state variables. The module also has a category and instance number.

```
void sendMixMatchSparts(void) {
  /* get the widget parameters */
  for (each program in the can's program list)
     for (each module in each category)
        call getParameters function;
  /* handle the programs */
  while (not done) {
     get a program from the can that has a non-empty spart pool;
     if (no such program exists)
        we are done;
     else {
        for (each spart in the spart pool) {
           if (AVO persistence mode is number of sparts type)
              delete the AVOs in the working buffer;
           if (grid mode) {
              set region of interest;
            for (each module except posupdate and death cats)
                 call init function;
              while (all the incarnations of the spart are not dead) {
                 for (each module except posupdate and death cats) {
                    call behav function;
                    update stream indexes;
                    if (animate and AVO persistence mode is number of positions) {
                       draw the scene;
                       delete the AVOs in the working buffer;
                       }
                    }
                 }
              }
           else {
              get the initial position of the spart in all its streams;
              if (spart is alive) {
                for (each module in each category)
                   call init function;
                }
              while (spart is alive) {
                for (each module in each category)
                   call behav function;
                if (AVO persistence mode is number of positions) {
                   draw the scene;
                   delete the AVOs in the working buffer;
                   }
                }
              }
            remove the spart from the spart pool;
            }
        }
     }
  }
```

Figure 5.9: Pseudocode for the execution of composed sparts.

```
typedef struct _moduleType {
  Str           name;                             /* name of component */
  ModuleCat     cat;                              /* its category */
  int           instance;                         /* instance number */
  int           noInputs;                         /* number of inputs */
  InOut         inNames[MAX_MOD_INPUTS];          /* names of inputs */
  InOut         inputs[MAX_MOD_INPUTS];           /* types of inputs */
  int           noOutputs;                        /* number of outputs */
  Str           outNames[MAX_MOD_INPUTS];         /* names of outputs */
  InOut         outputs[MAX_MOD_OUTPUTS];         /* types of outputs */
  void          **inOuts;                         /* addresses */
  int           noInterVars;                      /* number of variables */
  InOut         interVarNames[MAX_MOD_INTERVARS]; /* their names */
  InOut         interVarTypes[MAX_MOD_INTERVARS]; /* their types */
  void          **interVars;                      /* their addresses */
  void          (* behav)(Module *mod);           /* behavior function */
  void          (* init)(Module *mod);            /* initialize */
  void          (* createForm)(Module *mod);      /* create the forms */
  void          (* setForm)(Module *mod);         /* set the forms */
  void          (* getParameters)(Module *mod);   /* get the parameters */
  Widget        *paramWdgt;                       /* parameter widgets */
  } Module;
```

Figure 5.10: Structure of a component.

A spart composition consists of specifying what modules to use and how these modules depend on each other. The dependency is specified by tying outputs to inputs. During the execution of the composed sparts the modules need to know where to read the inputs and where to write the outputs. This is achieved by the `Map` structure that appears as an element in the `Composed` structure. Figure 5.11 illustrates abstractly the simple memory scheme used for inter-component data transfer.

Each module allocates as many pointers as it has inputs and outputs in the buffer pointed to by the `inOuts` element. These pointers are then assigned values during the parsing of the composition. When an output field is encountered during parsing, if the connection name is not already on the map, space is allocated for that output data type. The `address` element of the map as well as the particular output element of the `inOut` buffer is made to point to the allocated space. When an input field is encountered, the connection name is looked up in the map, and the address field is assigned to the relevant field in the `inOut` buffer. In this way, during its execution, the module obtains the input value by merely dereferencing the relevant `inOut` buffer element.

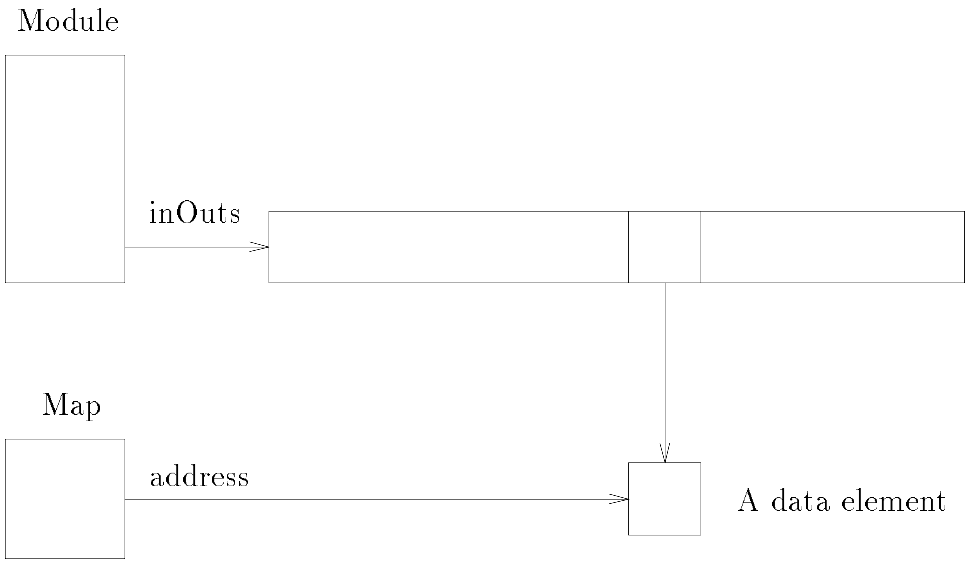


Figure 5.11: Diagram illustrating how a module looks up the address of an input field from the map structure

5.4 Component Writing

In *Mix&Match*, the user has to provide a top-level function called the *user* function that is the heart of the component. This function can be written as a normal C function whose arguments will hold the inputs, outputs, state variables and an array of parameters. In this case wrapper code is generated automatically by the configuration manager to provide the arguments to this function. If users would rather do away with this extra level of function call and work directly with some of the internals for efficiency, they can write a top-level function with one argument which is a pointer to either a `Module` or a `Predefined` structure.

Some components may provide a function that is to be executed once for each spart in the delivery as an initialization. Usually this would be the case if the component would like to keep an internal state and wishes to initialize that state variable.

Many components will require parameters that can be controlled through control widgets. If that is the case, there must be a function that creates the forms when the spart is loaded into a can. Users do not need to concern themselves with the body of this function as it is generated by the configuration manager. The control widgets may need a function that is user dependent to set some default values for the widgets. Users will need to provide a function for this purpose.

One final function that needs to be specified if the user function will depend on some parameters is a function that will be called at the beginning of a delivery once, for all the sparts. This function will call appropriate widget functions so as to stuff the parameters array that the top-level user function needs. This way of accessing parameters was chosen over the alternative of the user function accessing the control widgets each time since some of these calls can be expensive and components will likely execute many times during the delivery. In the chosen way, parameters are obtained once at the beginning and the user function accesses them by merely indexing into an array of parameters.

The data types that *Mix&Match* allows as inputs and outputs are as follows:

- *Byte.* This is equivalent to unsigned char.
- *Short.*
- *Int.*
- *Long.*
- *Float.*
- *Double.*
- *Boolean.* This is equivalent to an integer.
- *Stream.* The stream data type.
- *Vector.* A vector of three floats.
- *String.* An array of chars.
- *Object.* The geometry data type.

Inputs are passed by value to the user function except those that are structures such as `stream,vector,string` and `object` which are passed by reference.

5.4.1 Configuration Manager

The component writer uses the Configuration Manager (CM) to make the specifics of a component known to the system (see section 4.6). The CM uses this information to generate a component definition file. There may be up to two more files that are created. If the component writer has requested wrapper code or if the function has a control panel, then a file contains generated code. One function acts as a wrapper function for the top-level user function and supplies the arguments to it (figure 5.12). This hides some of the internal structure from the user. A second function is for the creation of the control panel holding the parameter widgets (figure 5.13).

```
/******************************************************/
/*                                                    */
/* The top-level function called by spray             */
/*                                                    */
/******************************************************/
void
uf_IsoSurf(Module *mod)
{
 StreamList *Stream;
 Bool Doit;
 int Index;
 float Threshold;
 GeoObject *Surface;

 /* get inputs */
 Stream = (StreamList *)(*(mod->inOuts));
 Doit = *((Bool *)(*(mod->inOuts+1)));
 Index = *((int *)(*(mod->inOuts+2)));
 Threshold = *((float *)(*(mod->inOuts+3)));

 /* get outputs */
 Surface = (GeoObject *)(*(mod->inOuts+4));

 /* call to the user function */
 IsoSurf(Stream, Doit, Index, Threshold, Surface, mod->paramWdgt->params);
}
```

Figure 5.12: Example wrapper code generated by the configuration manager. The details of the internal structure are thus hidden from the user who merely writes the function IsoSurf with the arguments based on the inputs, outputs and state variables.

```
/*****************************************************/
/*                                                   */
/* The create forms function.                        */
/*                                                   */
/*****************************************************/
void
cf_IsoSurf(Module *mod)
{
  FL_OBJECT *obj;

  mod->paramWdgt->form = fl_bgn_form(FL_NO_BOX,300.0,240.0);
  obj = fl_add_box(FL_UP_BOX,0.0,0.0,300.0,240.0,"");
  strcpy(mod->paramWdgt->paramNames[0], "Name");
  mod->paramWdgt->paramObjs[0] = obj =
    fl_add_box(FL_FRAME_BOX,50.0,170.0,200.0, 40.0,"");
  strcpy(mod->paramWdgt->paramNames[1], "TranspSl");
  mod->paramWdgt->paramObjs[1] = obj =
       fl_add_slider(FL_HOR_SLIDER,20.0,100.0,26 0.0,30.0,"Transparency");
    fl_set_object_align(obj,FL_ALIGN_TOP);
    fl_set_call_back(obj,isoSurfSliderVal, (long)mod);
  strcpy(mod->paramWdgt->paramNames[2], "TranspV");
  mod->paramWdgt->paramObjs[2] = obj =
       fl_add_input(FL_NORMAL_INPUT,110.0,80.0,8 0.0,20.0,"");
    fl_set_object_boxtype(obj,FL_FRAME_BOX);
    fl_set_object_color(obj,9,9);
  obj = fl_add_text(FL_NORMAL_TEXT,20.0,80.0,30.0,20.0,"0");
  obj = fl_add_text(FL_NORMAL_TEXT,250.0,80.0,30.0,20.0,"1");
  strcpy(mod->paramWdgt->paramNames[3], "ColCh");
  mod->paramWdgt->paramObjs[3] = obj =
       fl_add_choice(FL_NORMAL_CHOICE,70.0,20.0, 160.0,30.0,"Color");
    fl_set_object_boxtype(obj,FL_SHADOW_BOX);
    fl_set_object_align(obj,FL_ALIGN_TOP);
    fl_set_call_back(obj,isoSurfColChoiceCB, (long)mod);
  fl_end_form();
}
```

Figure 5.13: Example control panel creation code generated by the configuration manager. Objects have been given names dependent on the component generating it so that multiple instances can exist.

5.4.2 Application Programmer Interface (API)

There are a number of APIs for the use of the component writer. These are libraries that provide the means for a new component to access the system-specific data structures. Some are merely convenience functions to hide some of the structural detail from the user, others facilitate AVO creation and data interpolation. These functions are listed in appendix A. Here, the different APIs and their purposes are summarized:

- **Stream API.** This API provides the means to access the stream data structure. Perhaps the most often needed operation by a component is the data value at a certain spatial location. There are functions that return the scalar or the vector value at a given location. These use bilinear and trilinear interpolation in the case of regular and rectilinear 2D and 3D structured grids respectively. Since the the current implementation of *Mix&Match* is focused on meteorological data, transformations take place on the fly, if necessary, to locate the point in its native coordinate system. Other convenience functions provide a means to access members of the stream data structure.
- **Geometry API.** The *Visual* behavior components will normally output visualization objects that the renderer can render. There are a set of these objects that are available for use by the component writer including point sets, line sets, triangular meshes, and polygon sets. The geometry API consists of functions that define the objects and functions that set their attributes such as color and transparency. The component writer calls one of the geometry definition functions to create a set of geometric primitives. These get attached to the AVO list of the spray can from which the sparts containing the component have been delivered. Any attribute setting functions called apply to the most recent object defined. Attributes can be set for the whole object or for portions of the object. For instance, the color of a polygon object can be set such that a single color applies to the whole object. Alternatively, a different color can be supplied for each face or for each vertex.
- **Module API.** These are mostly convenience functions that enable the component writer to access input and output variables and parameter objects of components (modules).
- **Predefined Spart API.** Similarly, these are mostly convenience functions that enable the component writer to access input and output variables and parameter objects of predefined sparts.
- **Can API.** This API provides functions to obtain or set spray can properties.

- **Miscellaneous API.** There are other utility functions that are available such as safe memory allocation functions and error message reporting functions.

5.5 Performance

The particle nature of spray rendering and the fine granularity of the components make efficiency a concern. Essentially, the components making up the program must be executed at each location during the life-cycle of a spart. Although each of these components is usually quite simple, the overhead of calling these functions repeatedly can cause inefficiencies.

Another source of inefficiency is that the components producing visualization objects must operate locally. Consider the vector visualization technique of streamlines. A predefined spart could collect each point along the path of the spart and request a line-set object consisting of those points. An *a priori* estimate and allocation of storage could save on the number of system calls made for memory allocation. A *Mix&Match* spart that accomplishes the same task, on the other hand, would consist of a component that outputs a line segment at the current location and a component that updates the current location. Hence, the streamline is produced in piecemeal fashion as many small line-set objects rather than a single line-set object. There are three problems with this:

1. The rendering time of the scene suffers as the number of objects increases because of the overhead of traversing the list. The memory requirement for this representation is also higher.
2. The internal points making up the streamline are repeated, causing a doubling of storage required.
3. The visual component works independently and requires a system call to memory allocation each time it outputs a line segment.

A solution for the first problem is to gather objects with similar attributes into a single object. This is called *object compaction*, and the scheme is illustrated in figure 5.14. The compaction takes place after each delivery. Only the objects generated during a delivery are compacted automatically. The user may also request the compaction of the objects in the scene from a menu. The time required to do the compaction is more than offset by the time saved for rendering the uncompacted scene. Memory savings can also be enormous.

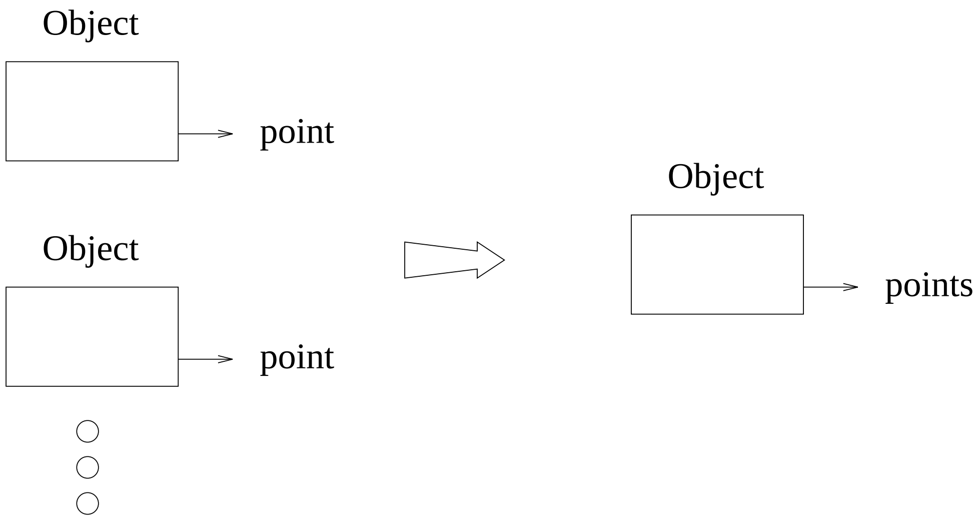


Figure 5.14: To improve rendering speed, objects with similar attributes are gathered into a single object. The compaction shortens the list of objects to be rendered.

Table 5.1, shows the results of an experiment on object compaction on some AVOs. The first and second columns of the table list the type and number of primitives in a scene while the third column shows the time to compact the scene in seconds. The *Before* and *After* columns list the number of AVOs, the total memory occupied by the list of AVOs in megabytes and the time to draw the scene in seconds before and after compaction has taken place[1]. The primitives are points, lines and polygons. The distinction between colored and monochromatic primitives is that in the former, there is a color associated with each primitive, whereas in the latter, a single color applies to all primitives within the object.

Table 5.2 summarizes the results of the experiment by showing the ratio of pre-compaction to post-compaction values for memory usage and rendering time for the cases with the highest number of primitives in table 5.1. It can be seen that the memory and time savings are considerable. For instance, non-compacted colored points are about 4 times as

[1] The experiment was carried out on an Onyx Reality Engine running on a single CPU.

Primitive		Time To	Before			After		
Type	Num(K)	Comp(s)	AVOs(K)	Mem(M)	Draw(s)	AVOs	Mem(M)	Draw(s)
Colored points	8.00	0.09	8.00	0.768	0.10	8	0.193	0.01
	40.00	0.49	40.00	3.840	0.50	40	0.963	0.03
	80.00	1.00	80.00	7.680	1.00	80	1.926	0.06
Mono points	8.00	0.10	8.00	0.768	0.10	8	0.097	0.01
	40.00	0.45	40.00	3.840	0.50	40	0.483	0.02
	80.00	0.92	80.00	7.680	1.00	80	0.967	0.03
Colored lines	8.00	0.16	8.00	1.006	0.13	24	0.482	0.02
	40.00	0.78	40.00	5.279	0.63	120	2.408	0.10
	79.99	1.60	79.99	10.559	1.26	240	4.817	0.19
Mono lines	8.00	0.15	8.00	0.960	0.12	24	0.290	0.01
	40.00	0.70	40.00	4.799	0.61	120	1.450	0.06
	79.99	1.42	79.99	9.599	1.18	240	2.900	0.13
Colored polys	0.89	0.02	0.89	0.227	0.03	6	0.163	0.01
	4.45	0.14	4.45	1.133	0.13	26	0.814	0.05
	8.90	0.26	8.90	2.265	0.27	51	1.628	0.10
Mono polys	0.89	0.02	0.89	0.190	0.02	6	0.116	0.01
	4.45	0.13	4.45	0.949	0.11	26	0.577	0.02
	8.90	0.25	8.90	1.898	0.22	51	1.154	0.04

Table 5.1: This table shows the results of an experiment on object compaction. The number of AVOs, memory and the time to draw the AVOs are compared before and after object compaction. Object compaction saves memory and results in faster rendering times after an initial cost.

Primitive	Memory	Time to draw
Colored points	3.99	16.7
Mono points	7.94	33.3
Colored lines	2.19	6.6
Mono lines	3.31	9.0
Colored polys	1.39	2.7
Mono polys	1.64	5.5

Table 5.2: This table summarizes the results of the experiment on object compaction. The ratio of pre-compaction to post-compaction values of memory and rendering time are give. Object compaction saves memory and results in faster rendering times.

big and may require about 17 times as long to render. The memory and rendering time savings for monochromatic points are even better. The time to compact a list of AVOs is about the same as the time to render them as uncompacted.

Data set	Spart	Primitives	AVOs	Memory(M)	Execution(s)
Sphere	ColMapPoint	8000	8000	0.768	0.18
	Dust	8000	1	0.192	0.04
Temperature	ColMapPoint	143008	143008	13.729	3.43
	Dust	143008	1	3.432	0.85
Sphere	IsoSurf	1090	890	0.190	0.17
	PreIsoSurf	1090	1	0.115	0.06
Temperature	IsoSurf	9184	8854	1.791	2.84
	PreIsoSurf	9184	1	1.047	0.65

Table 5.3: This table compares predefined sparts to composed sparts. *ColMapPoint* and *IsoSurf* are composed sparts while *Dust* and *PreIsoSurf* are their predefined counterparts. Predefined sparts are faster to execute and produce compact AVOs requiring less memory.

The compaction routines could be altered to also solve the second problem mentioned above (the repeating of internal streamline points and corresponding storage expansion). As the objects are gathered, repeated points could be ignored. However, this would require a search to see whether a new point already exists each time a new object is integrated. This is too costly to justify the saving of memory.

Another experiment was conducted to compare composed sparts to predefined sparts that basically accomplished the same task. The composed sparts *ColMapPoint* and *IsoSurf* were compared to the predefined sparts *Dust* and *PreIsoSurf* respectively. The *ColMapPoint* spart consists of two components while the *IsoSurf* spart contained three components (see section 6.2). The sparts were tried on two data sets: the sphere data set was of dimensions 20 x 20 x 20 while the NORAPS temperature data was 109 x 82 x 16. The times are given in seconds for the generation of the scene by grid traversal and does not include compaction and rendering times. The predefined sparts are faster in execution and produce already compact scenes that require less memory (table 5.3).

6. Components, Compositions and Visualizations

One of the design goals for this environment was functionality. For this purpose, many standard visualization techniques have been implemented in *Mix&Match*. Some techniques, such as streamlines, are inherently particle based and naturally map to the spray rendering paradigm. Others, such as the marching cubes iso-surface generation algorithm, are not particle based but can be adapted to also work in the spray rendering framework. This chapter lists the components that have been implemented and presents some sample compositions and visualizations that use them.

6.1 Sample Components

In designing components, it is important to bear in mind that there is a tradeoff between flexibility and efficiency. Usually, the finer in granularity and the more general the component is, the more flexibly it can be used in compositions. However, this implies that there are more components that make up a construction and hence more functions to be executed at each location. Another efficiency concern is to avoid repetition of compute intensive tasks in components. If a component calculates some value at some expense which might be useful for another component, it is desirable to output such values so that a component receiving them as input does not repeat the calculation. However, this makes the receiving component more dependent on others and restricts its flexibility.

In the following, the components that have been implemented are listed by category. These were implemented by breaking down a visualization technique into relevant components that fit the spray rendering paradigm. This decomposition allows the components to be used in other compositions.

6.1.1 Target Components

Target components are feature detection components, and output a boolean if a certain condition is satisfied. Only a few target components have been implemented so far. These are as follows:

- *IsoThresh.* This component takes a stream as input and returns true if the current spart position is in a cell that would produce an iso-value surface as in the marching cubes algorithm. The component also outputs the index to the case table as well as the iso-value being sought. Parameters that can be set are the iso-value and an option that performs automatic iso-value selection.
- *Counter.* This component is useful as a counter. The number of times the component has been executed is used in a relation to determine whether the target condition has been satisfied. The count and the relational operators are parameters that can be specified.
- *GetMagnitude.* This component is an example of a derived stream. The input is a vector stream and the output is a scalar stream that is the magnitude of the input.
- *Or.* Some logical operators have been implemented as target components so that the latter can be logically combined. This component outputs the logical OR of two boolean inputs.
- *And.* Outputs the logical AND of two boolean inputs.
- *Not.* Outputs the logical inverse of a boolean input.

6.1.2 Visual Components

Visual components are at the heart of the visualization technique. They usually accept a boolean as input and output some visualization objects if the condition is satisfied. The following visual components have been implemented.

- *OrthoSlice.* This component outputs a slice that is orthogonal to an axis. The slice is output as a grid object and is invisible. The inputs are a stream and a boolean and the output is a geometry type. Parameters specify the resolution of the grid, the axis for the slice and whether the slice should be along the axis of the current projection or the original projection.

- *RubberSheet.* This component takes a surface, and performs a displacement along the normal direction to the surface. It takes as input a stream, a boolean and a geometry and outputs the geometry. Parameters specify the range and scale of the displacement.
- *ColMapPoint.* This component places color mapped points at the current spart location.
- *ColMapSeg.* This component places a line segment from the current location to the previous location. The first time it is called, no geometry is defined and the current location is saved to be used later.
- *ColMapSph.* This component outputs a colored sphere at the current location. Parameters specify whether the color is a chosen color or whether it is data dependent. The size and transparencies of the spheres can also be specified.
- *IsoSurf.* This component works in conjunction with *IsoThresh* to generate portions of an iso-surface in the cell the current location is in. The transparency and the color can be specified as parameters. Normals can be inverted if desired.
- *AddColSurf.* This component takes geometry as input and maps a constant or stream dependent color to it.
- *ValText.* This component outputs the value at the current location as text. The text has 3D coordinates.
- *VecGlyph.* This component outputs 3D vector glyphs in the form of a cylinder capped with a cone. The color of the glyphs can be constant or dependent on the vector magnitude or some other scalar stream. The size can also be scaled depending on the vector magnitude. Another parameter acts as a filter so that only every nth call to the function outputs a glyph.
- *SpartView.* This component updates the camera position and direction. The current location of the spart becomes the camera position and the vector from the previous location to the current location specifies the camera direction. Field of view can be specified as a parameter. It is used for fly-by effects.

- *Contour.* This component takes as input a surface and outputs contour lines on the surface. Parameters specify the spacing between the contour lines, the line thickness and whether the lines should be a single color or data dependent.
- *Annotate.* This component can be used to drop annotations at locations. The annotation consists of a 3D arrow and text and various parameters specify the length and color of the text and the arrow. In a certain mode of operation, the text scrolls allowing cans to talk to each other in a collaborative setting.
- *BumpMap.* This component implements various techniques for vector visualization using bump-mapping[PA95].

6.1.3 Position Components

These components update the current location of the spart. Some are deterministic while others use a pseudo random number generator to achieve some nondeterministic behavior. Still others are data dependent.

- *BallPos.* This component places the current spart at the position of the ball that is tethered to the can. The ball is used for can manipulation and this component can be used to probe what is at the end of it. The component does not take any inputs or outputs.
- *StepAlongRay.* This component steps a certain distance along the initial direction of the spart. The distance can be factored and randomized by parameters.
- *RandBiDir.* This component places the spart randomly in either direction along the line segment that forms the intersection of the initial direction of the spart with the bounding box of the input stream. The spart will never leave the bounding box. Therefore, an appropriate death function needs to be included in the composition to make sure that the spart dies.
- *RandWalk.* This component, while traveling generally in the initial direction of the spart, jitters the position randomly.

- *VecInteg.* This is a data dependent position update function. Given a location in a vector stream, it will do an integration step to determine the new location. The integration step, type and direction can be specified through parameters.

6.1.4 Death Components

These components determine whether a spart should die or whether new ones should be spawned. There is a default death function that kills the spart once it gets out of the bounding box of the volume.

- *Conditional.* This component takes a boolean as input and kills the spart if it happens to be true.
- *Iteration.* This component has a counter that keeps track of the number of times the component has been executed. Once the target count is reached, the spart is killed.
- *Spawn.* This component spawns new sparts. The component takes as input a boolean condition that needs to be satisfied before the spawning can take place. Another input specifies the name of the spart that will be spawned. Thus, a spart can clone itself or mutate into other sparts. Parameters specify how many new sparts will be spawned and the range of directions they have initially.

6.2 Sample Compositions

In this section, some example compositions are given that use the components described in the previous section. These composed sparts are simple constructions. Yet, since it is possible to use multiple cans containing sparts multiple times, complex visualizations can be obtained. The sample compositions are illustrative. They can be easily changed and combined by including other components. The compositions are listed by name, the composition as it might appear in the textual editor and a short description of the spart. The images show their use in isolation.

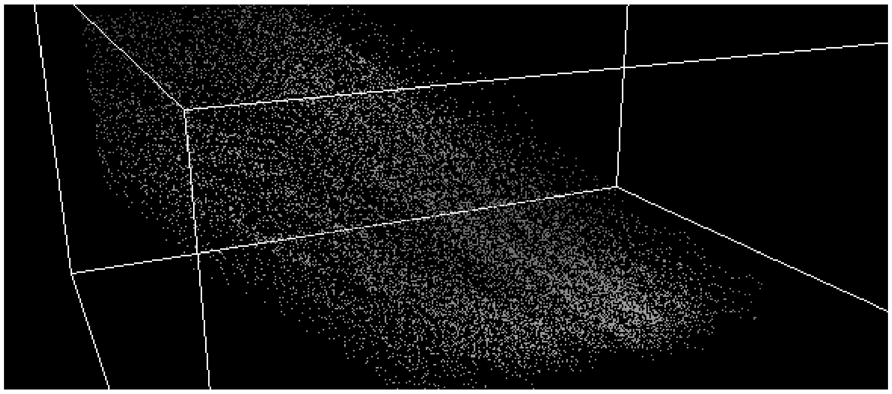


Figure 6.1: $\boxed{\textit{ColMapPoint}}$

```
ColMapPoint [ S1 ] [ =1 ]
RandWalk [ S1 ]
```

This spart simply maps the data value of the input stream to a colored point. The position update component allows some jittering to avoid regularity. The boolean input to the visual function is constant indicating that it does not depend on a target function.

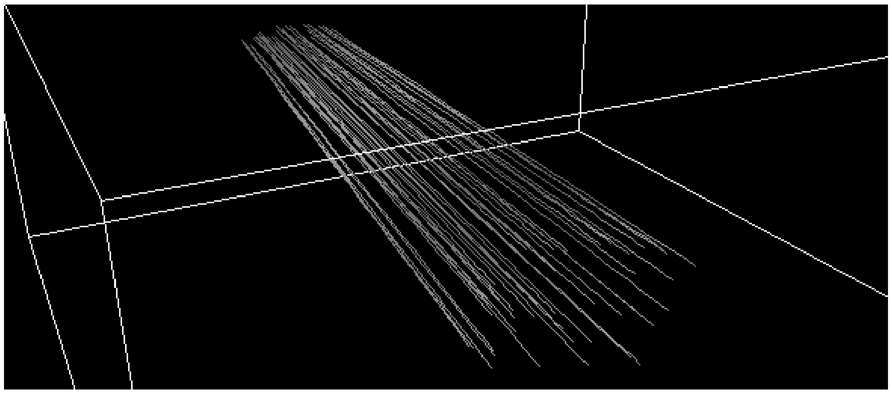


Figure 6.2: $\boxed{\textit{ColMapSeg}}$

```
ColMapSeg [ S1 ] [ =1 ]
StepAlongRay [ S1 ]
```

This spart outputs colored straight lines. In effect, lines dissect the data and map the values to color. Changing the position function to `RandWalk` would create crooked lines.

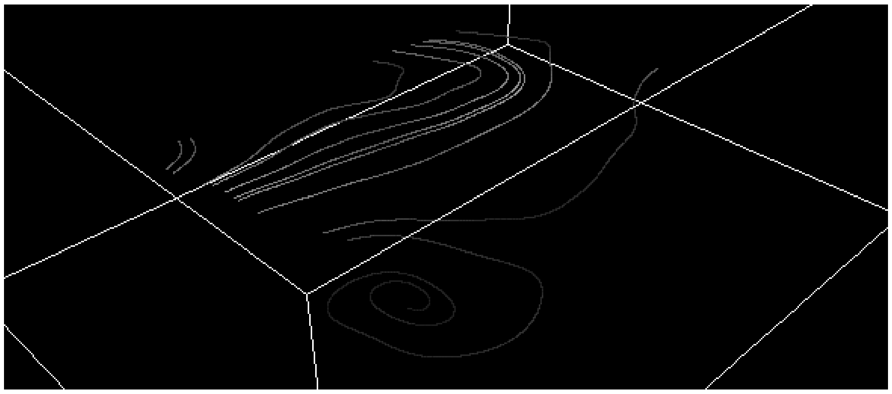


Figure 6.3: $\boxed{\textit{StreamLine}}$

```
ColMapSeg [ S1 ] [ =1 ]
VecInteg [ S1 ]
```

Replacing the position function with the data dependent `VecInteg` results in a streamline spart, a classic vector visualization technique.

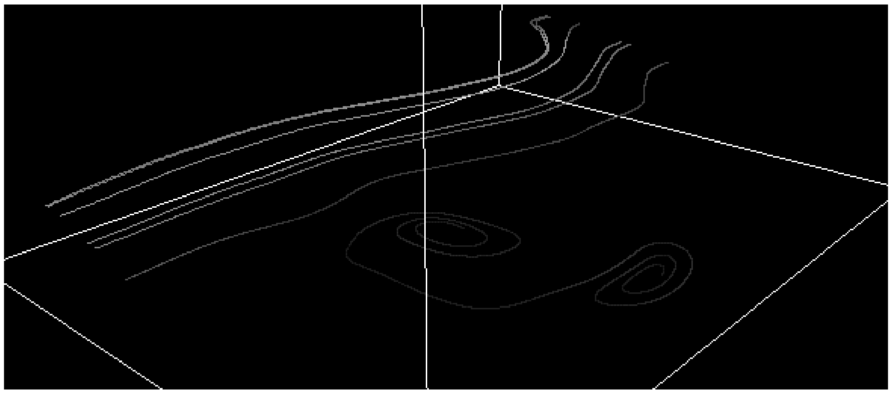


Figure 6.4: $\boxed{\textit{StrForwAndBack}}$

```
Counter ( Count )
ColMapSeg [ S1 ] [ =1 ]
VecInteg [ S1 ]
Spawn [ =StreamLine ] [ Count ]
```

The *StreamLine* spart only does forward integration. This spart uses the `Spawn` component to also do backward integration. The `Counter` component is used to launch the StreamLine spart while the `ColMapSeg` and `VecInteg` together form another streamline spart. Setting the parameter on the `VecInteg` to do the backward integration achieves the goal.

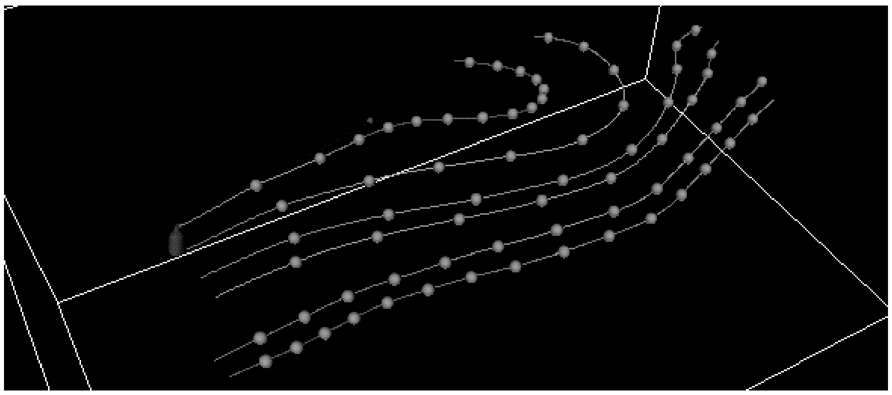


Figure 6.5: $\boxed{\textit{StrLineAndSph}}$

```
Counter ( Doit )
ColMapSeg [ S1 ] [ =1 ]
ColMapSph [ S2 ] [ Doit ]
VecInteg [ S1 ]
```

By including an extra visual component and a target component, one can obtain streamlines with spheres placed along them. The spheres can act on a separate stream so that two streams can be correlated.

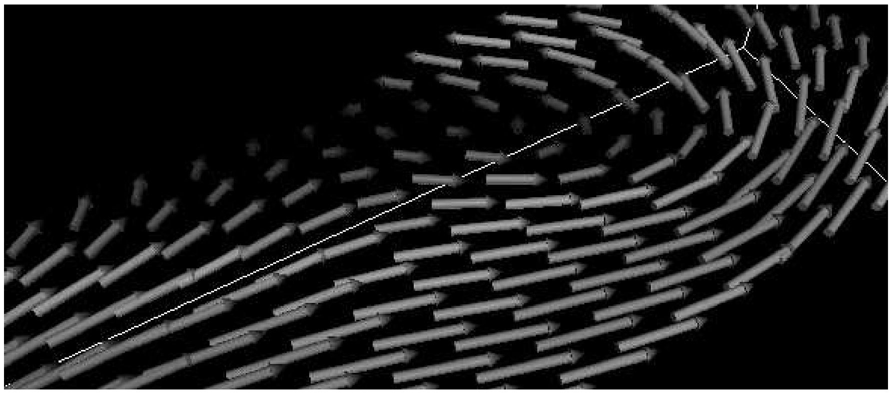


Figure 6.6: $\boxed{\textit{VecGlyph}}$

```
VecGlyph [ S1 ] [ S2 ] [ =1 ]
StepAlongRay [ S1 ]
```

This spart outputs vector glyphs based on a vector field. The glyphs are cylinders capped with cones and their lengths depend on the vector magnitude. They can be colored based on another stream.

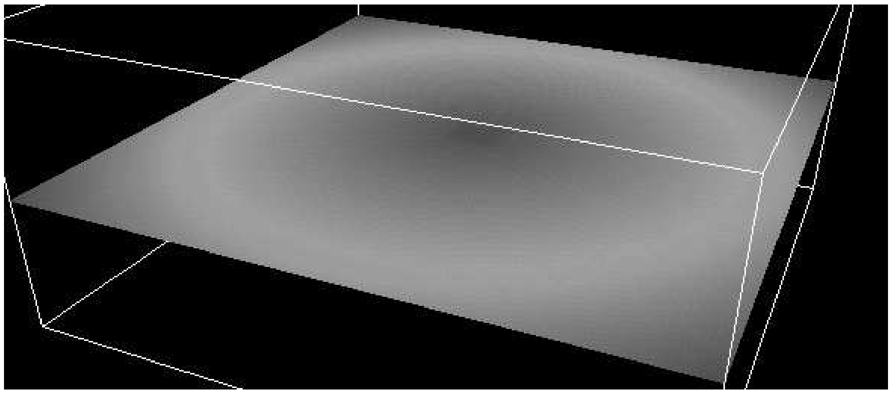


Figure 6.7: $\boxed{\textit{ColMapSlice}}$

```
OrthoSlice [ S1 ] [ =1 ] ( Slice )
AddColSurf [ S1 ] [ =1 ] [ Slice ] ( OBJECT )
Conditional [ =1 ]
```

This spart creates a color mapped slice orthogonal to one of the axes. The slice is placed at the current spart location and the spart dies in the first iteration since the `Conditional` death function is set to true.

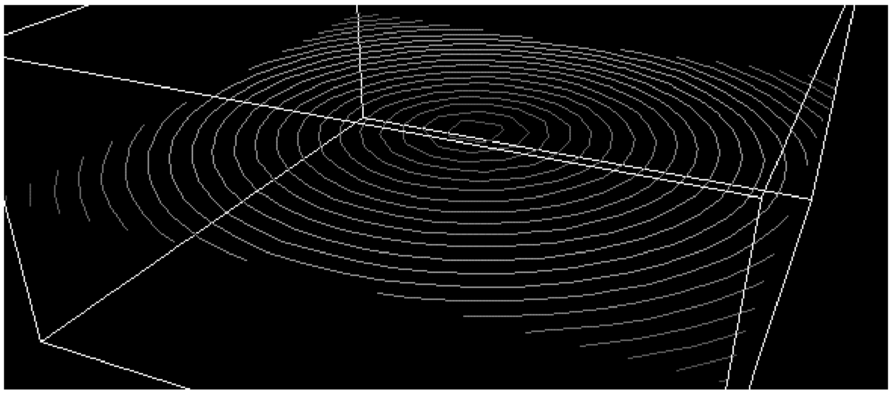


Figure 6.8: $\boxed{\textit{ContourSlice}}$

```
OrthoSlice [ S1 ] [ =1 ] ( Slice )
Contour [ S1 ] [ =1 ] [ Slice ] ( OBJECT )
Conditional [ =1 ]
```

Instead of pseudo colored slices, changing the visual component results in contour lines instead.

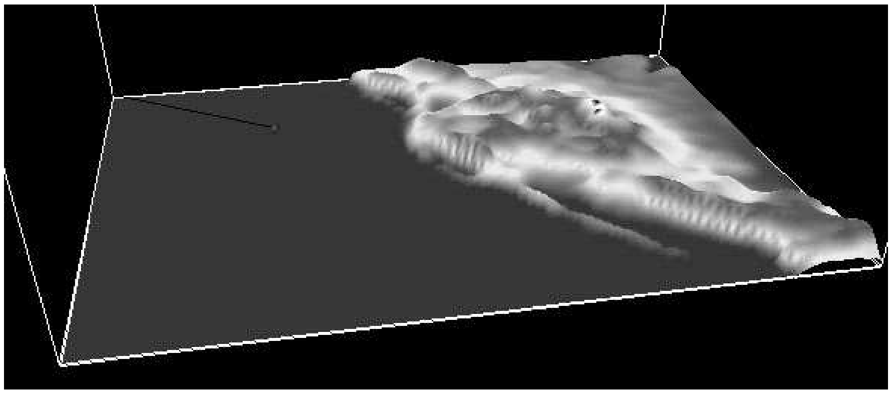


Figure 6.9: $\boxed{\textit{ColMapRubberSh}}$

```
OrthoSlice [ S1 ] [ =1 ] ( Slice1 )
AddColSurf [ S1 ] [ =1 ] [ Slice1 ] ( Slice2 )
RubberSheet [ S2 ] [ =1 ] [ Slice2 ] ( OBJECT )
Conditional [ =1 ]
```

Adding an extra visual to *ColMapSlice* allows rubber sheeting. In other words, the grid nodes are displaced by an amount scaled by the value of a second stream. Correlation of the two streams is thus possible.

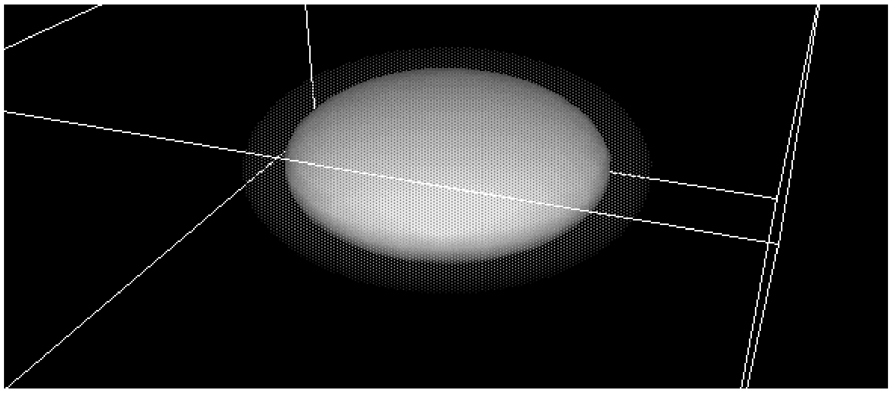


Figure 6.10: $\boxed{\textit{IsoSurf}}$

```
IsoThresh [ S1 ] ( Found ) ( Index ) ( IsoVal )
IsoSurf [ S1 ] [ Found ] [ Index ] [ IsoVal ] ( OBJECT )
StepAlongRay [ S1 ]
```

The iso-surface spart has a target function that detects the existence of a surface. The visual component generates a polygonal surface.

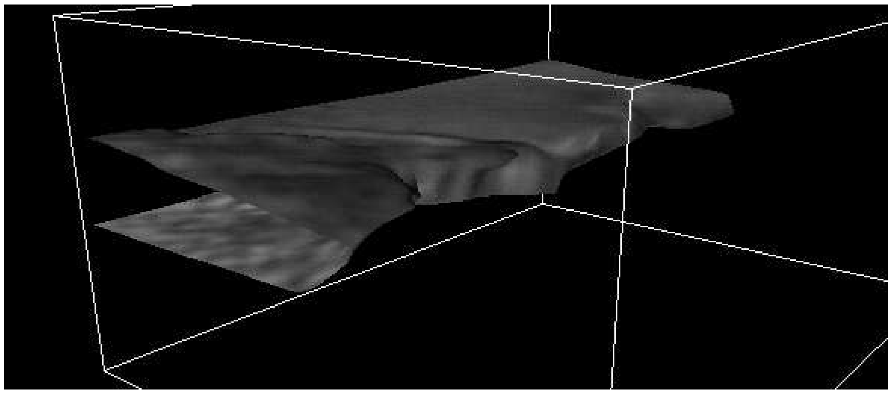


Figure 6.11: $\boxed{\textit{ColMapSurf}}$

```
IsoThresh [ S1 ] ( Found ) ( Index ) ( IsoVal )
IsoSurf [ S1 ] [ Found ] [ Index ] [ IsoVal ] ( Surface )
AddColSurf [ S2 ] [ Found ] [ Surface ] ( OBJECT )
StepAlongRay [ S1 ]
```

Adding the `AddColSurf` component that depends on a second stream adds color to the surface based on the second stream.

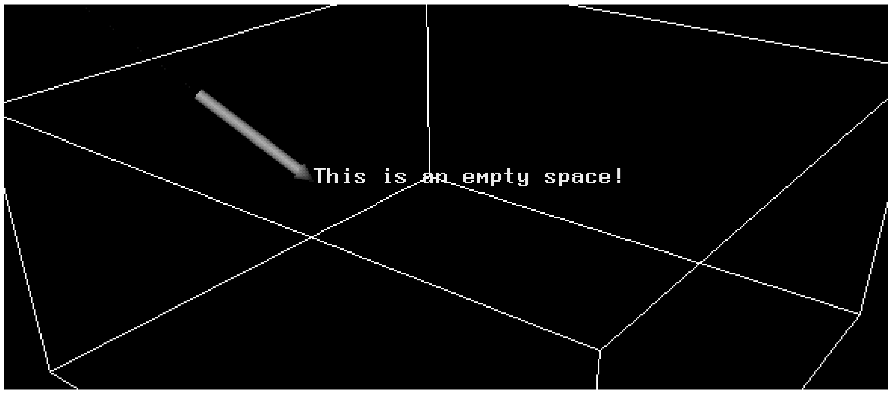


Figure 6.12: $\boxed{\textit{Annotate}}$

```
Annotate [ S1 ] [ =1 ]
Conditional [ =1 ]
```

The *Annotate* spart has been defined so that it outputs the text at the can's ball position and immediately dies.

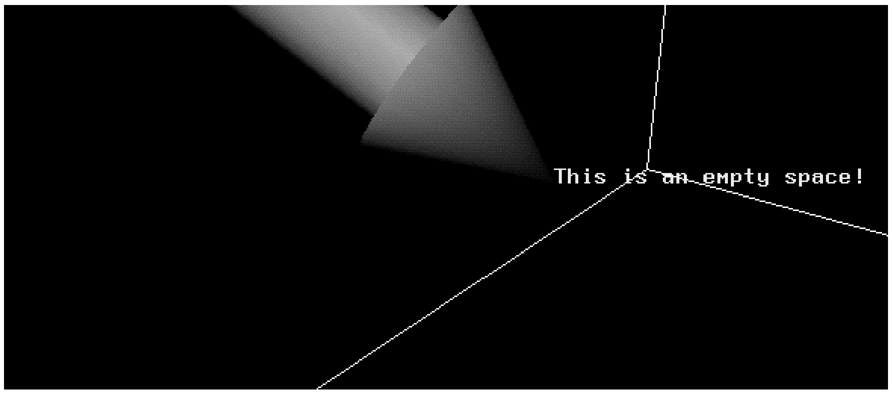


Figure 6.13: $\boxed{\textit{SpartView}}$

```
SpartView [ S1 ] [ =1 ]
StepAlongRay [ S1 ]
```

This spart achieves a fly-through effect. The `SpartView` component updates the camera position based on the current and previous locations of the spart. If the position update component is replaced with the `VecInteg` component, we could achieve the effect of flying along a streamline.

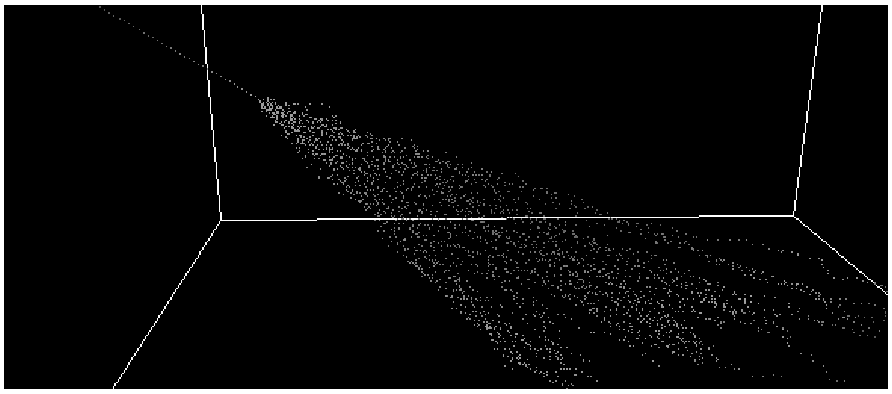


Figure 6.14: $\boxed{\textit{ColMapPointSpawn}}$

```
Counter ( Count )
ColMapPoint [ S1 ] [ =1 ]
StepAlongRay [ S1 ]
Conditional [ Count ]
Spawn [ =ColMapPoint ] [ Count ]
```

This spart gives another example of the use of the `Spawn` component. The parent spart is killed and other *ColMapPoint* sparts are launched instead.

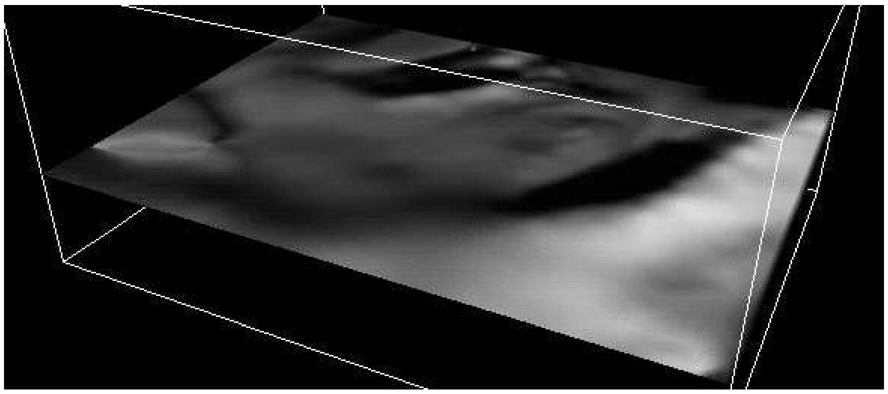


Figure 6.15: $\boxed{\textit{SliceBumpMap}}$

```
OrthoSlice [ S1 ] [ =1 ] ( Slice1 )
AddColSurf [ S1 ] [ =1 ] ( Slice1 ) ( Slice2 )
BumpMap [ S1 ] [ =1 ] ( Slice2 ) ( OBJECT )
Conditional [ =1 ]
```

This spart applies various bump-mapping techniques for vector field visualization. In the example above, the magnitude of the vector field has been used as the perturbation function while the direction is mapped to an HSV color wheel.

6.3 Sample Visualizations

In the previous section, some sample compositions were presented and images of their application in isolation were shown. The user can create a spray can containing these sparts and use it as a tool to apply the visualization technique multiple times. Multiple instances of a tool can also exist and may operate on different data. More complex visualizations can thus be obtained by using multiple spray cans multiple times. The images in this section provide a few examples of such visualizations.

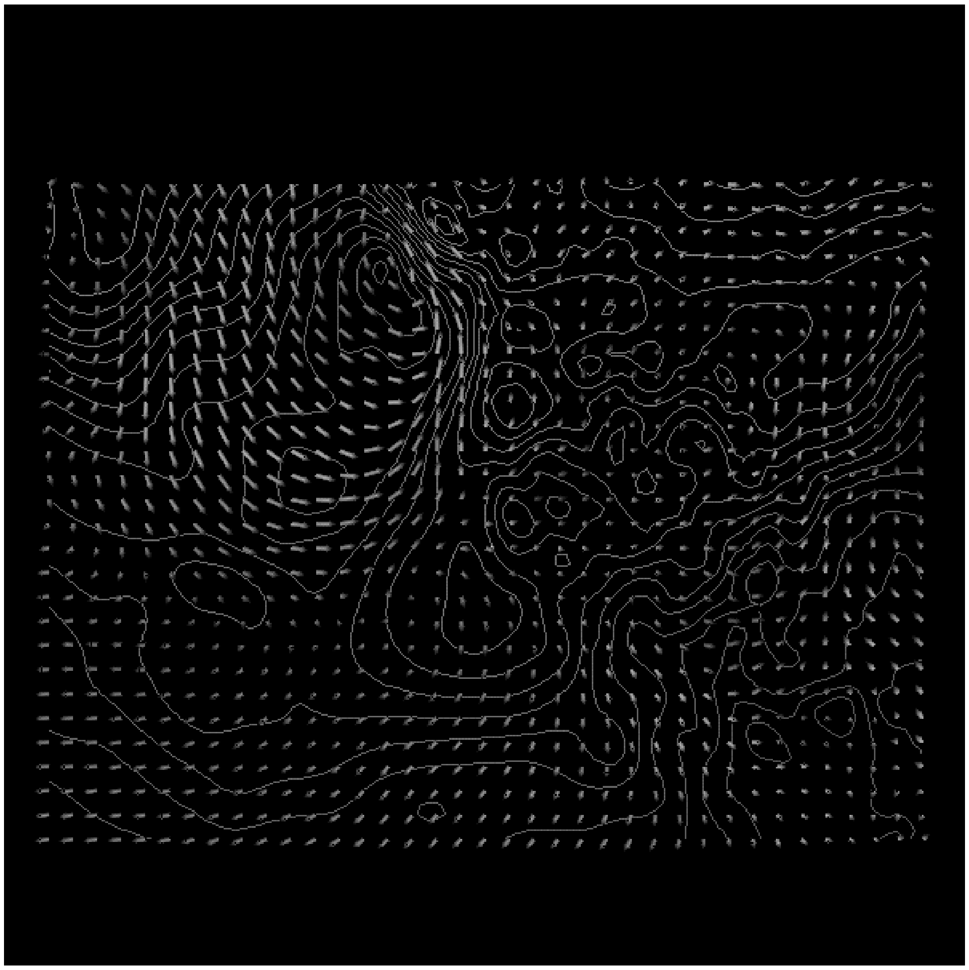


Figure 6.16: In this visualization, the *ContourSlice* spart has been applied to a slice of a 3D temperature field of the NORAPS climate model data. On the same slice, the *VecGlyph* spart has been applied to the wind field with subsamplings along the x and y axes. The colors of the vector glyphs have been mapped to the relative humidity field. Hence, three different fields are being correlated.

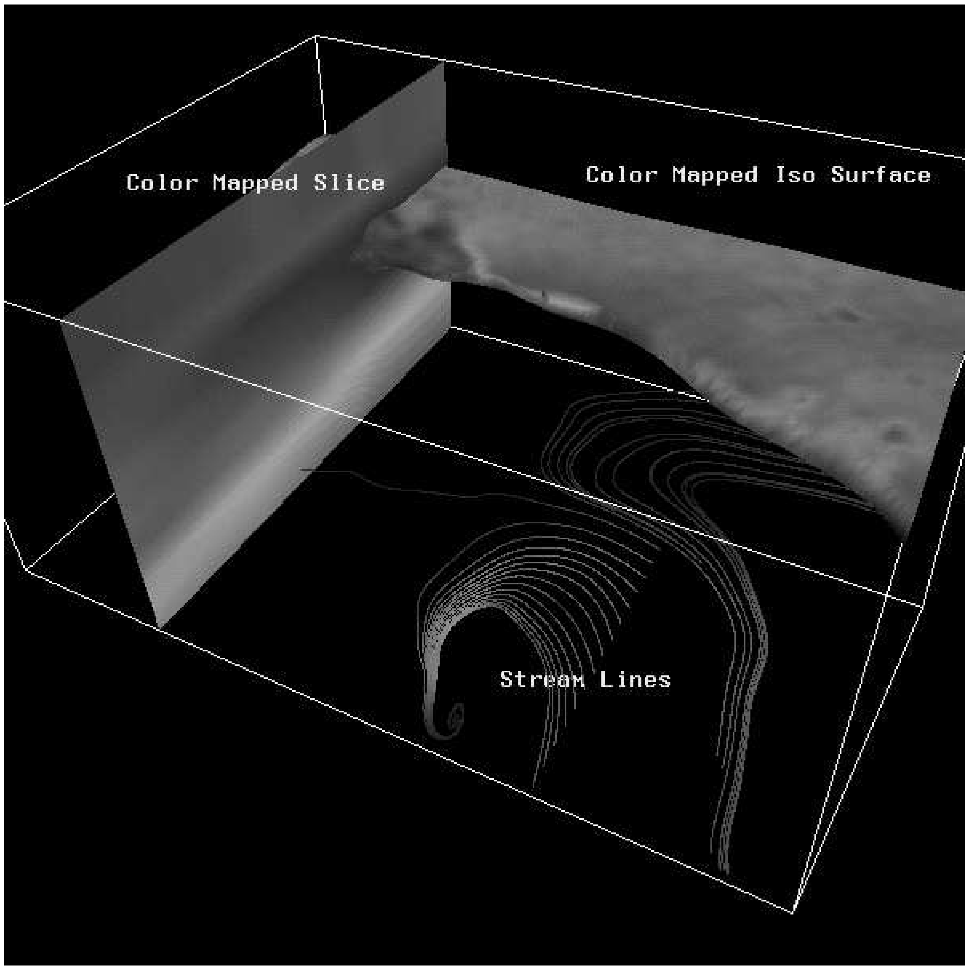


Figure 6.17: Here, the *ColMapSlice* spart has been used to get a vertical pseudocolor slice of the temperature field. The *ColMapSurf* spart has been applied to get the distribution of relative humidity over the iso-valued temperature surface. Some streamlines applied to the wind field by *StreamLine* spart are also shown. The text is the result of multiple applications of the *Annotate* spart.

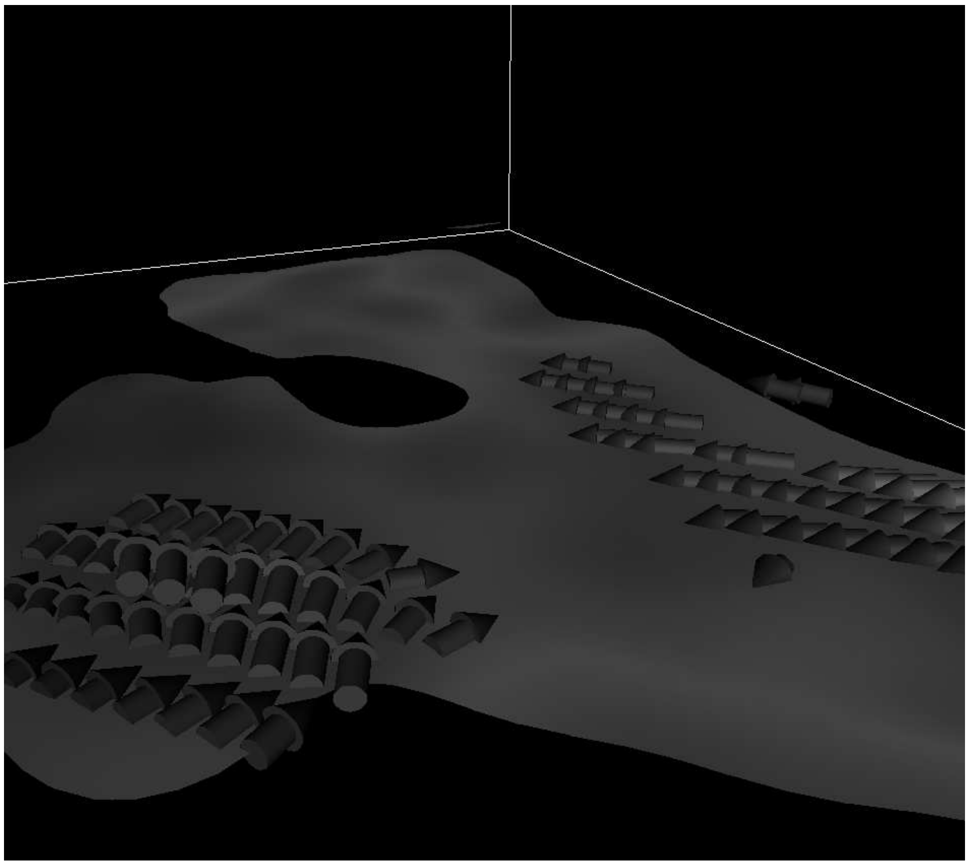


Figure 6.18: This is an example of a single application of a more complicated spart. An isosurface is generated on the geopotential field and the relative humidity is colormapped over it. Vector glyphs of the wind field are placed at positions where the temperature field would have produced an iso-valued surface.

7. Conclusions and Future Work

A new, tool-oriented scientific visualization environment has been presented. *Mix&Match* is a modular and extensible system that borrows from both data-flow-based MVEs and spray rendering. The spray can and smart particle metaphors of spray rendering are used to model fine grained components that can be used to compose the overall behavior of a spart. The visual programming approach of MVEs is used for the composition of sparts.

The *Mix&Match* environment offers three advantages over data flow based MVEs: direct interactivity, greater flexibility and easier extensibility. Direct interaction helps the scientist in exploring the data set that is being analyzed. Flexibility permits more tools to be constructed from a given number of components and benefits the tool constructor who wishes to experiment with new tools. Easier extensibility encourages component writers to add to the system and try out new visualization techniques.

Direct interaction is achieved by using the spray can metaphor of spray rendering. The use of the spray can as a tool for all techniques allows high level mode settings to be used that result in a variety of interactions without changing compositions. Scientists can use the same tool in different ways and gain insight into their data sets.

Although the particle nature of the paradigm and the repeated execution of small components can lead to memory and processor time inefficiencies, interactivity is still achieved by using the spray can as a probe. An object compaction scheme aids in this process by saving memory and rendering time.

The fine granularity of the components allows greater flexibility in affecting what happens at the local level. This, in turn, allows more varied compositions from a set of components and results in a richer collection of techniques. Many visualization techniques only differ in detail and share many common features. Instead of a module encapsulating the whole visualization technique, the technique can be broken down into simpler components which can be used in other constructions. In the design and implementation of *Mix&Match*, many of the popular visualization techniques were decomposed to work in this fashion. A simple, particle based template was used for this purpose which provides a unifying representation for the visualization techniques.

The fine granularity of components and the simple template also result in easier extensibility. The conceptual simplicity of small components that perform local and specific tasks permits new ones to be readily added to the system encouraging the exploration of new visualization techniques. This task is further facilitated by a configuration manager that presents a graphical interface for component integration.

Mix&Match was also successfully used to experiment with new visualization techniques. Of note, is the use of bump mapping to visualize vector data[PA95].

There is much room for the extension of *Mix&Match* features. Scripting would allow batch processing at the expense of interactivity. This would help users who want to just see an end product at the press of a button. The performance could be improved with a new design of the geometry API and a memory management mechanism. The support of time-varying and scattered data sets would be very valuable. Allowing sparts to communicate may allow novel visualization techniques to be realized. The addition of virtual reality capabilities would make the 3D spray can manipulations more intuitive. What is more intriguing and worth exploring is whether the concepts of sparts composed of components of fine granularity are more amenable to parallelization and simulation steering.

Appendix A. Application Programmer Interface(API)

A.1 Stream API

A.1.1 Macros

```
#define spStrGetTitle(s)                  ((s)->title)
#define spStrGetCoordSys(s)               ((s)->coordSys)
#define spStrNeedToConvert(s)             ((s)->needToConcert)
#define spStrGetType(s)                   ((s)->strType)
#define spStrInUse(s)                     ((s)->inUse)
#define spStrSpartIsDead(s)               ((s)->spartIsDead)
#define spStrStructGetNumDims(s)          ((s)->t.lat.nDim)
#define spStrStructGetDims(s)             ((s)->t.lat.dims)
#define spStrStructGetXYSize(s)           ((s)->t.lat.xysize)
#define spStrStructGetCurIndexes(s)       ((s)->t.lat.curIndexes)
#define spStrStructGetLowIndexes(s)       ((s)->t.lat.lows)
#define spStrStructGetHighIndexes(s)      ((s)->t.lat.highs)
#define spStrStructGetSubs(s)             ((s)->t.lat.subs)
#define spStrStructGetData(s)             ((s)->t.lat.data)
#define spStrStructGetCoords(s)           ((s)->t.lat.coords)
#define spStrStructGetNumData(s)          ((s)->t.lat.data.nDataVar)
#define spStrStructGetMin(s)              ((s)->t.lat.data.min)
#define spStrStructGetMax(s)              ((s)->t.lat.data.max)
#define spStrStructGetRaw(s)              ((s)->t.lat.data.raw)
#define spStrStructGetScaled(s)           ((s)->t.lat.data.scaled)
#define spStrStructGetCoordType(s)        ((s)->t.lat.coords.type)
#define spStrStructGetCurMinCellSides(s) ((s)->t.lat.coords.minCSides)
#define spStrStructGetCurMinCellSide(s)  ((s)->t.lat.coords.minCSide)
#define spStrStructGetOrMinCellSides(s)  ((s)->t.lat.coords.orMinCSides)
#define spStrStructGetOrMinCellSide(s)   ((s)->t.lat.coords.orMinCSide)
#define spStrStructGetEpsilon(s)          ((s)->t.lat.coords.epsilon)
#define spStrStructGetCurSides(s)         ((s)->t.lat.coords.sides)
#define spStrStructGetCurBBLow(s)         ((s)->t.lat.coords.BBl)
#define spStrStructGetCurBBHigh(s)        ((s)->t.lat.coords.BBh)
#define spStrStructGetOrSides(s)          ((s)->t.lat.coords.orSides)
#define spStrStructGetOrBBLow(s)          ((s)->t.lat.coords.orBBl)
#define spStrStructGetOrBBHigh(s)         ((s)->t.lat.coords.orBBh)
#define spStrGetScalarValAtNode(d, i)     (d)[(i)]
#define spStrGetScaledVectorValAtNode(d, i)  (d)[(i)]
#define INDEX(s, x, y, z)                 \
        ((x) + (y)*s->t.lat.dims[X] + (z)*s->t.lat.xysize)
#define INDEX_2D(s, x, y)                 ((x) + (y)*s->t.lat.dims[X])
#define spStrKillSpart(s)                 \
        ((s)->spartIsDead = CurSpart.isDead = TRUE)
```

A.1.2 Functions

```
Bool spStrGetPosInOriginal(StreamList *stream, Vertex pos, Vertex npos);
Bool spStrGetPosInCurrent(StreamList *stream, Vertex pos, Vertex npos);
Bool spStrGetScalarValAtCurPos(StreamList *stream, float *data, Vertex pos,
                            float *value);
Bool spStrGetVectorValAtCurPos(StreamList *stream, float *data,
                            Vertex pos, Vertex npos, Vertex value);
void spStrGetVectorValAtNode(StreamList *stream, long index, Vertex value);
Bool spStrPosInOriginal(StreamList *stream, Vertex pt);
Bool spStrPosInCurrent(StreamList *stream, Vertex pt);
void spStrSetBallRegion(StreamList *stream, CanPtr c, Vertex ballPos,
                            int size);
void spStrSetRegionOfInterest(StreamList *stream, CanPtr c);
void spStrSetWholeOfVolume(StreamList *stream, CanPtr c);
Bool spStrGetScalarValAtOrPos(StreamList *stream, float *data, Vertex pos,
                            float *value);
Bool spStrGetVectorValAtOrPos(StreamList *stream, float *data, Vertex pos,
                            Vertex vec);
Bool spStrGetIndexFromOrPos(StreamList *stream, Vertex loc, int indices[]);
Bool spStrGetIndexFromCurPos(StreamList *stream, Vertex loc,
                            int indices[]);
Bool spStrGetOrPosFromIndex(StreamList *stream, int indices[], Vertex loc);
Bool spStrGetCurPosFromIndex(StreamList *stream, int indices[],
                            Vertex loc);
Bool spStrGetPOModelCell(StreamList *stream, Vertex pos, int *i,
                            int *j, int *k);
```

A.2 Geometry API

A.2.1 Functions

```
void spGeoBeginDefine(ObjList *list)}
void spGeoEndDefine(ObjList *list)
void spGeoGeoPointsDefine(int n, Vertex *point)
void spGeoGeoLinesDefine(int np, Vertex *point, int ni, long *index,
                            short lineWidth)
void spGeoGeoPolysDefine(int np, Vertex *point, int ni, long *index)
void spGeoGeoTrisDefine(int np, Vertex *point, int ni, long *index)
void spGeoGeoSpheresDefine(int n, Vertex *point, float *radius)
void spGeoGeoCylindersDefine(int n, Vertex *pnt0, Vertex *pnt1,
                            float *radius)
void spGeoGeoConesDefine(int n, Vertex *pnt0, Vertex *pnt1, float *radius)
void spGeoGeoDisksUpDefine(int n, Vertex *point0, Vertex *point1,
                            float *radius)
void spGeoGeoDisksDownDefine(int n, Vertex *point0, Vertex *point1,
                            float *radius)
void spGeoGeoGridDefine(int nu, int nv, Vertex *point)
void spGeoGeoTextDefine(int n, Vertex *point, char *text)
void spGeoGeoNormalAdd(int n, Vertex *normal, GeoPer per)
void spGeoGeoColorAdd(int n, Color *color, GeoPer per)
void spGeoGeoTransparencyAdd(int n, float *transp, GeoPer per)
```

A.3 Module API

A.3.1 Macros

```
#define spModSetInterVar(i, t, v)      \
        (*((t *)(*(mod->interVars+(i)))) = (v))
#define spModGetRefInput(i, t)         ((t *)(*(mod->inOuts+(i))))
#define spModGetInput(i, t)            (*((t *)(*(mod->inOuts+(i)))))
#define spModGetParamObj(i)            mod->paramWdgt->paramObjs[(i)]
#define spModGetParamObjFromArg(a, i)  \
        ((Module *)a)->paramWdgt->paramObjs[(i)]
#define spModGetIntVal(i)              mod->paramWdgt->intVals[(i)]
#define spModSetIntVal(i, v)           (mod->paramWdgt->intVals[(i)] = (v))
#define spModGetIntValFromArg(a, i)    \
        ((Module *)a)->paramWdgt->intVals[(i)]
#define spModSetIntValFromArg(a, i, v) \
        (((Module *)a)->paramWdgt->intVals[(i)] = (v))
```

A.3.2 Functions

```
int spModGetInputIndex(Module *mod, char *name);
int spModGetInputIndexArg(long a, char *name);
int spModGetOutputIndex(Module *mod, char *name);
int spModGetOutputIndexArg(long a, char *name);
int spModGetInterVarsIndex(Module *mod, char *name);
int spModGetInterVarsIndexArg(long a, char *name);
int spModGetParamIndex(Module *mod, char *name);
int spModGetParamIndexArg(long a, char *name);
```

A.4 Predefined API

A.4.1 Macros

```
#define spPreInterVarExists(i)         (*(predef->interVars+(i)) != NULL)
#define spPreCreateInterVar(i, t)      (*(predef->interVars+(i)) = NEW(t))
```

```
#define spPreGetInterVars(i)          (*(predef->interVars+(i)))
#define spPreSetInterVar(i, t, v)     \
         (*((t *)(*(predef->interVars+(i)))) = (v))
#define spPreGetParam(i)              predef->paramWdgt->params[(i)]
#define spPreSetParam(i, v)           \
         (predef->paramWdgt->params[(i)] = (v))
#define spPreGetInput(i)              predef->streams[i]
#define spPreGetParamObj(i)           predef->paramWdgt->paramObjs[i]
#define spPreGetParamObjFromArg(a, i)  \
         ((Predefined *)a)->paramWdgt->paramObjs[i]
#define spPreGetIntVal(i)             predef->paramWdgt->intVals[i]
#define spPreSetIntVal(i, v)          \
         (predef->paramWdgt->intVals[(i)] = (v))
#define spPreGetIntValFromArg(a, i)    \
         ((Predefined *)a)->paramWdgt->intVals[i]
#define spPreSetIntValFromArg(a, i, v) \
         (((Predefined *)a)->paramWdgt->intVals[i] = (v))
```

A.4.2 Functions

```
int spPreGetInputIndex(Predefined *predef, char *name);
int spPreGetInputIndexArg(long a, char *name);
int spPreGetInterVarsndex(Predefined *predef, char *name);
int spPreGetInterVarsndexArg(long a, char *name);
int spPreGetParamIndex(Predefined *predef, char *name);
int spPreGetParamIndexArg(long a, char *name);
```

A.5 Can API

A.5.1 Macros

```
#define spCanGetPosition()      (CurrentCan->pos)
#define spCanGetDirection()     (CurrentCan->dir)
#define spCanGetDistToBall()    (CurrentCan->CanCenterPointDist)
#define spCanGetBallPos(b)      FIND_POS((b), CurrentCan->pos, \
          CurrentCan->dir, \
          CurrentCan->CanCenterPointDist)
#define spCanGetPrograms()      (CurrentCan->programs)
#define spCanGetColMap()        (CurrentCan->colMap)
#define spCanGetModeSpartDel()  (CurrentCan->sprayMode.spartDel)
#define spCanGetModeDensity()   (CurrentCan->sprayMode.spray.density)
#define spCanGetModeNozSize()   (CurrentCan->sprayMode.spray.nozzleSize)
#define spCanGetModeNozShape()  (CurrentCan->sprayMode.spray.nozzleShape)
#define spCanGetModeRegType()   (CurrentCan->sprayMode.grid.regionType)
#define spCanGetModeMins()      (CurrentCan->sprayMode.grid.mins)
```

```
#define spCanGetModeMaxs()      (CurrentCan->sprayMode.grid.maxs)
#define spCanGetModeSubs()      (CurrentCan->sprayMode.grid.subs)
#define spCanGetModeAnims()     (CurrentCan->sprayMode.grid.anims)
#define spCanGetModeRegSize()   (CurrentCan->sprayMode.grid.regionsize)
#define spCanGetAVOPers()       (CurrentCan->sprayMode.AVOPersist)
```

A.6 Miscellaneous API

A.6.1 Macros

```
#define MAXM(x, y)               (((x) > (y)) ? (x) : (y))
#define MINM(x, y)               (((x) < (y)) ? (x) : (y))
#define IS_ODD( a )              ((a) & 0x1)
#define IS_EVEN( a )             (!((a) & 0x1))
#define SQR(a)                   ((a)*(a))
#define NEW(t)                   (t *)memAlloc(1, sizeof(t))
#define VEC_CP(a, b)             \
         (a)[X] = (b)[X]; (a)[Y] = (b)[Y]; (a)[Z] = (b)[Z]
#define VEC_CP_2D(a, b)          (a)[X] = (b)[X]; (a)[Y] = (b)[Y]
#define VEC_MUL_2D(a, b)         (b)[X]*=(a); (b)[Y]*=(a)
#define VEC_LEN_2D(a)            (sqrtf(SQR((a)[X]) + SQR((a)[Y])))
#define VEC_SET(v, x, y, z)      (v)[X]=x; (v)[Y]=y; (v)[Z]=z
#define VEC_SUB(a, b, c)         (c)[X] = (a)[X]-(b)[X];\
         (c)[Y] = (a)[Y]-(b)[Y];\
         (c)[Z] = (a)[Z]-(b)[Z]
#define VEC_SUM(a, b, c)         (c)[X] = (a)[X]+(b)[X];\
         (c)[Y] = (a)[Y]+(b)[Y];\
         (c)[Z] = (a)[Z]+(b)[Z]
#define VEC_CROSS(a, b, c)       (c)[X] = (a)[Y]*(b)[Z] - (a)[Z]*(b)[Y];\
         (c)[Y] = (a)[Z]*(b)[X] - (a)[X]*(b)[Z];\
         (c)[Z] = (a)[X]*(b)[Y] - (a)[Y]*(b)[X]
#define VEC_DOT(a, b)            \
 ((a)[X]*(b)[X] + (a)[Y]*(b)[Y] + (a)[Z]*(b)[Z])
#define VEC_LEN(a)               \
         (sqrtf(SQR((a)[X]) + SQR((a)[Y]) + SQR((a)[Z])))
#define VEC_MUL(a, b)            (b)[X]*=(a); (b)[Y]*=(a); (b)[Z]*=(a)
#define VEC_DIV(a, b)            (b)[X]/=(a); (b)[Y]/=(a); (b)[Z]/=(a)
#define VEC_SAME(a, b)           (((a)[X] == (b)[X]) && \
         ((a)[Y] == (b)[Y]) && \
         ((a)[Z] == (b)[Z]))
#define GET_DIST(a, b)           (sqrtf(SQR((b)[X]-(a)[X]) + \
         SQR((b)[Y]-(a)[Y]) + \
         SQR((b)[Z]-(a)[Z])))
#define FIND_POS(pos, b, u, t)   (pos)[X] = (b)[X] + ((t)*((u)[X]));\
         (pos)[Y] = (b)[Y] + ((t)*((u)[Y]));\
         (pos)[Z] = (b)[Z] + ((t)*((u)[Z]))
```

A.6.2 Functions

```
char *memAlloc(size_t nelem, size_t elsize);
char *memRealloc(char *place, size_t size);
void showMessage(char *str1, char *str2, char *str3);
```

A.7 Other Material

A user guide for the novice and intermediate users of *Mix&Match* exists. A separate guide for the component writer is also available. An mpeg movie showing example interactions is available under /projects/onr/mixmatch/video.

References

[A$^+$92] J. Almond et al. Visualization '91 workshop report: Scientific visualization environments. *Computer Graphics*, 26(3):213 – 216, 1992.

[AHH$^+$94] Ricardo S. Avila, Taosong He, Lichan Hong, Arie. E. Kaufman, Hanspeter Pfister, Claudio Silva, Lisa M. Sobierajski, and Sidney Wang. VolVis: A diversified volume visualization system. In *Proceedings: Visualization '94*, pages 31 – 38. IEEE Computer Society, 1994.

[Amk89] S. Amkraut. Flock: A behavioral model for computer animation. Master's thesis, Ohio State University, 1989. Art Education.

[ASK92] Ricardo S. Avila, Lisa M. Sobierajski, and Arie. E. Kaufman. Towards a comprehensive volume visualization system. In *Proceedings: Visualization '92*, pages 13 – 20. IEEE Computer Society, 1992.

[AVS92] AVS. *AVS Technical Overview*. Advanced Visual Systems Inc., Waltham, MA, 1992. Part Number 320-0018-02, Rev. A.

[BAWW90] D. L. Brittain, J. Aller, M. Wilson, and S. C. Wang. Design of an end-user data visualization system. In *Proceedings: Visualization '90*, pages 323 – 328. IEEE Computer Society, 1990.

[Bel87] R. G. Belie. Some advances in digital flow visualization. In *AIAA Aerospace Sciences Conference*, Reno, NV, January 1987. AIAA-87-1179.

[Bel93] A.J.C. Belien. Comparison of visualization techniques and packages. Technical report, Stichting Academisch Rekencentrum Amsterdam, Amsterdam, The Netherlands, 1993. ISBN 90-72490-08-8.

[Bli82a] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transaction on Graphics*, 1(3):235 – 256, 1982.

[Bli82b] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21 – 29, 1982.

[Blo88] J. Bloomenthal. Polygonization of implicit surfaces. *Computer-Aided Geometric Design*, 5:341 – 355, 1988.

[BMP$^+$90] G. V. Bancroft, F. J. Merritt, T. C. Plessel, P. G. Kelaita, R. K. McCabe, and A. Globus. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings: Visualization '90*, pages 14 – 27. IEEE Computer Society, 1990.

[BP89] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45 – 51, 1989.

[Bun88] P. G. Buning. Sources of error in the graphical analysis of CFD results. *J. Sci. Comp.*, 3(2), 1988.

[CA91] R. A. Crawfis and M. J. Allison. A scientific visualization synthesizer. In *Proceedings: Visualization '91*, pages 262 – 267. IEEE Computer Society, 1991.

[Cha90] J. Challinger. Object-oriented rendering of volumetric and geometric primitives. Master's thesis, University of California, Santa Cruz, CA, 1990.

[CL93] Brian Cabral and Leith(Casey) Leedom. Imaging vector fields using line integral convolution. *Computer Graphics (ACM Siggraph Proceedings)*, 27(4):263 – 270, August 1993.

[CLL$^+$88] H. E. ClineW., E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3), 1988.

[CM92] R. Crawfis and N. Max. Direct volume visualization of three-dimensional vector fields. *Proceedings of the Boston Workshop on Volume Visualization(ACM Siggraph)*, pages 55 – 60, 1992.

[CM93] Roger A. Crawfis and Nelson Max. Texture splats for 3d scalar and vector field visualization. In *Proceedings: Visualization '93*, pages 261 – 266. IEEE Computer Society, 1993.

[Dic89] R. R. Dickinson. A unified approach to the design of visualization software for the analysis of field problems. In *Three-Dimensional Visualization and Display Technologies*, volume 1083, pages 173 – 180. SPIE, 1989.

[dLvW93] Willem C. de Leeuw and Jarke J. van Wijk. A probe for local flow field visualization. In *Proceedings: Visualization '93*, pages 39 – 45. IEEE Computer Society, 1993.

[Dye90] D. S. Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60 – 69, 1990.

[Edw93] G. J. Edwards. The design of a second generation visualization environment. In J.J. Connor, S. Hernandez, T.K.S. Murthy, and H. Power, editors, *Visualization and Intelligent Design in Engineering and Architecture*, pages 3 – 16. Computational Mechanics Publications, Elsevier Science Publishers, 1993.

[EOR89] P. Eliasson, J. Oppelstrup, and A. Rizzi. STREAM 3D: Computer graphics program for streamline visualization. *Advances in Engineering Software*, 11(4), 1989.

[FGR85] G. Frieder, D. Gordon, and R.Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52 – 59, 1985.

[FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693 – 702, 1977.

[FL91] T. A. Foley and D. A. Lane. Multi-valued volumetric visualization. In *Proceedings: Visualization '91*, pages 218 – 225. IEEE Computer Society, 1991.

[For94] L. K. Forssell. Visualizing flow over curvilinear grid surfaces using line integral convolution. In *Proceedings: Visualization '94*, pages 240 – 247. IEEE Computer Society, 1994.

[FZY84] E. J. Farrell, R. Zappulla, and W.C. Yang. Color 3-d imaging of normal and pathologic intracranial structures. *IEEE Computer Graphics and Applications*, 4(9):5 – 17, September 1984.

[Gar90] M. P. Garrity. Raytracing irregular volume data. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):35 – 40, 1990.

[Gol86] S. Goldwasser. Rapid techniques for the display and manipulation of 3d biomedical data. *NCGA '86 Tutorial*, May 1986.

[Gou88] M. L. Gough. *NSSDC CDF Implementer's Guide(DEC VAX/VMS Version 1.1.* National Space Science Data Center, NASA/Goddard Space Flight Center, Greenbelt, MD, 1988.

[GR85] D. Gordon and R. A. Reynolds. Image space shading of 3-dimensional objects. *Computer Vision, Graphics and Image Processing*, 29:361 – 376, 1985.

[GW93] Allen Van Gelder and Jane Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *Proceedings: Visualization '93*, pages 70 – 77. IEEE Computer Society, 1993.

[HB86] K.H. Hoehne and R. Bernstein. Shading 3d-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, MI-5(1):45 – 47, March 1986.

[HD91] R. Haimes and D. Darmofal. Visualization in computational fluid dynamics: A case study. In *Proceedings: Visualization '91*, pages 392 – 397. IEEE Computer Society, 1991.

[HL79] 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.

[HLC91] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings: Visualization '91*, pages 298 – 305. IEEE Computer Society, 1991.

[HPvW94] Lambertus Hesselink, Frits H. Post, and Jarke J. van Wijk. Research issues in vector and tensor field visualization. In L. Rosenblum, R. A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann, editors, *Scientific Visualization: Advances and Challenges*, pages 488 – 495. Academic Press, London, 1994.

[HR92] J. P. M. Hultquist and E. L. Raible. SuperGlue: A programming environment for scientific visualization. In *Proceedings: Visualization '92*, pages 243 – 249. IEEE Computer Society, 1992.

[HS90] B. Hibbard and D. Santek. The Vis-5D system for easy interactive visualization. In *Proceedings: Visualization '90*, pages 28 – 35. IEEE Computer Society, 1990.

[Hul92] J. P. M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings: Visualization '92*, pages 171 – 177. IEEE Computer Society, 1992.

[KASS93] Peter Kochevar, Zahid Ahmed, Jonathan Shade, and Colin Sharp. Bridging the gap between visualization and data management: A simple visualization management system. In *Proceedings: Visualization '93*, pages 94 – 101. IEEE Computer Society, 1993.

[KH84] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(4):165 – 174, 1984.

[KM92] D. N. Kenwright and G. D. Mallinson. A 3-d streamline tracking algorithm using dual stream functions. In *Proceedings: Visualization '92*, pages 62 – 68. IEEE Computer Society, 1992.

[Kri91] R. D. Kriz. PV-Wave Point & Click. *Pixel*, pages 28 – 30, July/August 1991.

[Kru90] W. Krueger. Volume rendering and data feature enhancement. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):21 – 26, 1990.

[LAC+92] 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.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163 – 169, 1987.

[Leg91] S. M. Legensky. Advanced visualization on desktop workstations. In *Proceedings: Visualization '91*, pages 372 – 378. IEEE Computer Society, 1991.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29 – 37, 1988.

[Lev91] H. Levkowitz. Color icons: Merging color and texture perception for integrated visualization of multiple parameters. In *Proceedings: Visualization '91*, pages 164 – 170. IEEE Computer Society, 1991.

[Mat94] S. V. Matveyev. Approximation of isosurface in the marching cube: Ambiguity problem. In *Proceedings: Visualization '94*, pages 288 – 292. IEEE Computer Society, 1994.

[Max86] N. Max. Light diffusion through clouds and haze. *Computer Vision, Graphics and Image Processing*, 33(3):280 – 292, 1986.

[MBC93] Nelson Max, Barry Becker, and Roger Crawfis. Flow volume for interactive vector field visualization. In *Proceedings: Visualization '93*, pages 19 – 24. IEEE Computer Society, 1993.

[McC87] B. H. McCormick. Visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.

[Mea82] D. J. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129 – 147, 1982.

[Mer91] Philip J. Mercurio. The data visualizer. *Pixel*, pages 31 – 34, July/August 1991.

[MHC90] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):27 – 34, 1990.

[MP88] E. Murman and K. Powell. Trajectory integration in vortical flows. *AIAA Journal*, 27(7):982 – 984, 1988.

[MS90] C. Montani and R. Scopigno. Rendering volumetric data using the sticks representation scheme. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):87 – 93, 1990.

[MSS94] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *Proceedings: Visualization '94*, pages 281 – 287. IEEE Computer Society, 1994.

[Nat89] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. *NCSA HDF Calling Interfaces and Utilities. Version 3.0*, 1989.

[NH91] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings: Visualization '91*, pages 83 – 90. IEEE Computer Society, 1991.

[NSG90] K. L. Novins, F. X. Sillion, and D. P Greenberg. An efficient method for volume rendering using perspective projection. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):95 – 102, 1990.

[PA95] Alex Pang and Naim Alper. Bump mapped vector fields. In *Symposium on Electronic Imaging: Visual Data Exploration and Analysis*. SPIE, 1995.

[PD84] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 17(3):253 – 260, 1984.

[PS93a] A. Pang and K. Smith. Spray rendering: A new framework for visualization. Technical report, Board of Studies in C. I. S., University of California, Santa Cruz, CA 95064, 1993. UCSC-CRL-93-01.

[PS93b] Alex Pang and Kyle Smith. Spray rendering: Visualization using smart particles. In *Proceedings: Visualization '93*, pages 283 – 290. IEEE Computer Society, 1993.

[RD90] Russ Rew and Glenn P. Davis. NetCDF: An interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76 – 82, 1990.

[Ree83] W. T. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. *Computer Graphics*, 17(3):359 – 376, 1983.

[Rey87] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25 – 34, 1987.

[Sab86] M. Sabin. A survey of contouring methods. *Computer Graphics Forum*, 5:325 – 340, 1986.

[Sab88] P. Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics*, 22(4):51 – 58, 1988.

[SCN$^+$93] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, Alan Su, and Jiang Wu. Tioga: A database-oriented visualization tool. In *Proceedings: Visualization '93*, pages 86 – 93. IEEE Computer Society, 1993.

[SH94] Barton Stander and John Hart. A lipschitz method for accelerated volume rendering. In *Proceedings: 1994 Symposium on Volume Visualization*, pages 107 – 114. ACM SIGGRAPH, 1994.

[SK90] D. Speray and S. Kennan. Volume probes : Interactive data exploration on arbitrary grids. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):5 – 12, 1990.

[Slo92] G. Sloane. *IRIS Explorer Module Writer's Guide*. Silicon Graphics, Inc, Mountain View, 1992. Document Number 007-1369-010.

[Smi87] A. R. Smith. Volume graphics and volume visualization: A tutorial. Technical report, Pixar, San Rafael, CA, 1987. Technical Report 176.

[SSW86] D. S. Schlusselberg, W. K. Smith, and D. J. Woodward. Three-dimensional display of medical image volumes. *Proceedings of NCGA*, pages 114 – 123, 1986.

[ST90] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):63 – 70, 1990.

[SVL91] W. J. Schroeder, C. R. Volpe, and W. E. Lorensen. The Stream Polygon: A technique for 3D vector field visualization. In *Proceedings: Visualization '91*, pages 126 – 131. IEEE Computer Society, 1991.

[SZL92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics (ACM Siggraph Proceedings)*, 26(2):65 – 70, 1992.

[Tre93] Lyoyd A. Treinish. *Unifying Principles of Data Management for Scientific Visualziation.* Association for Computing Machinery, August 1993. SIGGRAPH '93 Course 02 Notes: Introduction to Scientific Visualization Tools and Techniques.

[TS87] Y. Trousset and F. Schmitt. Active-ray tracing for 3d medical imaging. *Proc. Eurographics '87*, pages 139 – 149, 1987.

[TT84] H. K. Tuy and L. T. Tuy. Direct 2d display of 3d objects. *IEEE Computer Graphics and Applications*, 4(10):29 – 33, 1984.

[Tur92] G. Turk. Re-tiling polygonal surfaces. *Computer Graphics (ACM Siggraph Proceedings)*, 26(2):55 – 64, 1992.

[UK88] C. Upson and M. Keeler. The V-buffer : Visible volume rendering. *Computer Graphics*, 22(4):59 – 64, 1988.

[Ups89] C. Upson. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30 – 42, 1989.

[Ups91] Craig Upson. Volumetric visualization techniques. In David F. Rogers and Rae A. Earnshaw, editors, *State of the Art in Computer Graphics*, pages 313 – 350. Springer-Verlag, New York, 1991.

[Vol89] G. Volpe. Streamlines and streamribbons in aerodynamics. Technical report, AIAA, 1989. Technical Report AIAA-89-0140.

[vW91] J. J. van Wijk. Spot Noise: Texture synthesis for data visualization. *Computer Graphics*, 25(4):309 – 318, 1991.

[vW92] J. J. van Wijk. Rendering surface particles. In *Proceedings: Visualization '92*, pages 54 – 61. IEEE Computer Society, 1992.

[vW93] Jarke J. van Wijk. Implicit stream surfaces. In *Proceedings: Visualization '93*, pages 245 – 252. IEEE Computer Society, 1993.

[WCA$^+$90] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct volume rendering of curvilinear volumes. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):41 – 46, 1990.

[Wes89] L. Westover. Interactive volume rendering. *Proceedings of the Chapel Hill Workshop on Volume Visualization(ACM Siggraph)*, pages 9 – 16, 1989.

[Wes90] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367 – 376, 1990.

[Wet90a] Mark Vande Wettering. apE 2.0. *Pixel*, pages 30 – 34, November/December 1990.

[Wet90b] Mark Vande Wettering. The application visualization system – AVS 2.0. *Pixel*, pages 30 – 33, July/August 1990.

[WMW86] B. Wyvill, C. McPheeters, and G. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227 – 234, 1986.

[WVG90a] J. Wilhelms and A Van Gelder. Octrees for faster isosurface generation. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):57 – 62, 1990.

[WVG90b] J. Wilhelms and A Van Gelder. Topological considerations in isosurface generation: Extended abstract. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):79 – 86, 1990.

[WVG91] J. Wilhelms and A Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics*, 25(4):275 – 284, 1991.

[YK92] R. Yagel and A. Kaufman. Template-based volume viewing. *EUROGRAPHICS '92*, 11(3):153 – 167, 1992.

[YP88] P. K. Yeung and S. B. Pope. An algorithm for tracking fluid particles in numerical simulations of homogeneous turbulence. *J. Comp. Physics*, 79:373, 1988.