THE SCIRUN PROBLEM SOLVING ENVIRONMENT AND COMPUTATIONAL STEERING SOFTWARE SYSTEM

by

Steven Gregory Parker

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

August 1999

Copyright © Steven Gregory Parker 1999

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Steven Gregory Parker

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Christopher R. Johnson

John B. Carter

Charles D. Hansen

James S. Painter

Peter S. Shirley

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Steven Gregory Parker in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Christopher R. Johnson
Chair, Supervisory Committee

Approved for the Major Department

Robert Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Since the introduction of computers, scientists and engineers have attempted to harness their power to simulate complex physical phenomena. Today, the computer is an almost universal tool used in a wide range of scientific and engineering domains.

Currently, organizing, running and visualizing a new large-scale simulation still requires hours or days of a researcher's time. Time and effort required for data input, output and conversion further slows and complicates process.

We present the design and application of SCIRun, a Problem Solving Environment (PSE), and a computational steering software system. SCIRun allows a scientist or engineer to interactively steer a computation, changing parameters, recomputing, and then revisualizing all within the same programming environment. The tightly integrated modular environment provided by SCIRun allows computational steering to be applied to a broad range of advanced scientific computations.

This dissertation demonstrates that computational steering can be a useful tool in computational science, engineering, and medicine applications and that this utility is obtained by providing a flexible and efficient infrastructure whereby disparate computational tools can be used in a single, focused environment. Furthermore, we demonstrate that using such an environment, a scientist or engineer can rapidly investigate the solution space for iterative computational design problems.

Four applications are demonstrated: Torso defibrillator modeling, Monte Carlo global illumination, a computational fluid dynamics application, and a simulation of atmospheric diffusion. These applications were selected to explore the gamut of a flexible and extensible problem solving environment.

CONTENTS

ABSTRACT . . . iv

LIST OF FIGURES . . . ix

ACKNOWLEDGEMENTS . . . xii

CHAPTERS

1. INTRODUCTION . . . 1

1.1 Motivation . . . 2
1.2 Computational Steering . . . 6
1.3 Problem Solving Environments . . . 7
1.3.1 Responsibilities of a Problem Solving Environment . . . 8
1.3.2 Requirements of the Application . . . 9
1.3.3 Implementing a PSE . . . 10
1.4 Thesis . . . 11
1.5 Evaluation of Thesis . . . 12
1.6 Contributions . . . 14
1.7 Thesis Organization . . . 14
1.8 Publications . . . 15

2. RELATED WORK . . . 17

2.1 Taxonomy of Steering Systems . . . 17
2.1.1 Previous Classifications . . . 18
2.1.2 Existing Tools for Steering . . . 19
2.1.2.1 Lightweight Steering: Scripting Languages and Wrappers . . . 19
2.1.2.2 CUMULVS . . . 19
2.1.2.3 Progress and Magellan . . . 20
2.1.2.4 VASE . . . 20
2.1.2.5 Pablo . . . 21
2.1.2.6 CSE . . . 21
2.1.3 A Taxonomy for Steering . . . 22
2.2 Visualization Systems . . . 22
2.2.1 AVS . . . 23
2.2.2 Iris Explorer . . . 24
2.2.3 Data Explorer . . . 24
2.2.4 vtk . . . 24
2.3 Chapter Summary . . . 24

3. SYSTEM OVERVIEW 25
3.1 Design Goals 25
3.2 Applications 26
3.3 SCIRun 28
3.4 Steering Optimizations 29
3.4.1 Data Structure Management 30
3.4.2 Progressive Refinement 31
3.4.3 Exploiting Interaction Coherence 31
3.5 Chapter Summary 32

4. DATAFLOW IMPLEMENTATION 33
4.1 Demand-Driven Dataflow 36
4.2 Flexible Dataflow Ports 38
4.2.1 Non-dataflow Ports 38
4.2.2 Fine-grained Dataflow 39
4.3 Steering in a Dataflow System 39
4.4 Chapter Summary 40

5. DATA MODELS 43
5.1 Common Datatypes 44
5.1.1 Matrix 45
5.1.2 The Mesh Class 46
5.1.3 Fields 47
5.1.4 Surfaces Data Structures 48
5.1.5 Other Datatypes 48
5.2 Persistent Data Storage 49
5.3 Chapter Summary 52

6. SUPPORT LIBRARIES 53
6.1 `Malloc, operator new`: libMalloc 54
6.2 The Multitask Library: libMultitask 56
6.2.1 Tasks 58
6.2.2 Intertask Communication 58
6.3 Generic Tools: libClasslib 59
6.3.1 TrivialAllocator 60
6.3.2 Handles and LockingHandles 61
6.3.3 Timers 61
6.4 Geometry Library 62
6.5 The Math Library 62
6.6 Chapter Summary 63

7. MODULES 64
7.1 Writing a Module 64
7.2 FEM and Matrix Modules 65
7.3 Readers and Writers 68
7.4 Visualization Modules 69

7.5 Salmon Module 72
7.6 Other Modules 73
7.7 Performance 74
7.8 Chapter Summary 75

8. RESULTS AND DISCUSSION 76

8.1 Application: Torso defibrillator modeling 76
8.1.1 Geometric Modeling 79
8.1.2 Numerical Analysis 80
8.1.3 Scientific Visualization 81
8.1.4 SCIRun Implementation 82
8.2 Application: Monte Carlo Global Illumination 87
8.2.1 SCIRun Implementation 88
8.3 Application: CFD Applications Using CFDLIB 91
8.3.1 SCIRun Implementation 92
8.3.2 Fortran Code 94
8.3.3 Results 95
8.4 Application: Atmospheric Dispersion 96
8.4.1 Spatial Discretization Method 98
8.4.2 Time Integration 98
8.4.3 Mesh Generation and Adaptation 100
8.4.4 Integration with SCIRun 101
8.4.5 Atmospheric Diffusion Simulation Results 102
8.5 Discussion 104
8.6 Comparison with Existing Systems 106
8.6.1 Performance 106
8.6.2 Flexibility 109
8.7 Chapter Summary 110

9. CONCLUSIONS 112

9.1 Contributions 113
9.2 Pros and cons of the SCIRun approach 115

10. ONGOING AND FUTURE WORK 118

10.1 Common Component Architecture 118
10.2 C-SAFE 119
10.3 Parallel CSPRINT/Tetrad 121
10.4 Neuroscience Inverse Problem 121
10.5 Inverse-EEG Pipeline 121
10.6 Optical Tomography 122
10.7 Visualization and Manipulation of
Large-scale Datasets 122
10.8 Local and Global Visualization Using
Haptic Devices 123
10.9 Chapter Summary 124

APPENDIX: SOFTWARE DESIGN 125

REFERENCES 130

LIST OF FIGURES

1.1 Example of the results of a computational process. This is a visualization of the electrical currents in the human torso, at one point in a normal heartbeat [56, 57, 77, 78, 106]. 5

1.2 Example of an interactive computational process. Unlike the example in Figure 1.1, this visualization is steerable - the user can change the placement of the electrodes, solution methods, and other numerical and geometrical parameters. 13

2.1 Taxonomy of steering systems and tools. The horizontal axis represents three different types of steering, and the vertical axis represents the method of interaction. The bubbles represent each of the systems' predominant strength. Arrows indicate portions of the space that are also covered by the system. 23

3.1 An example of a fairly complex dataflow network, showing the SCIRun modules (the boxes), the connections (the wires between them), and the input/output ports (the points on the modules that the wires connect). Magnified portions show simulation and modeling components (top left) connected to visualization components (bottom left) in a cohesive environment. 29

3.2 A closeup view of a dataflow network. A vector field, produced by the Gradient module, is consumed by both the Streamline and Hedgehog modules. In SCIRun, the data are shared between the modules so that the data do not need to be duplicated. 30

4.1 An example of a simple dataflow network, used to compute streamlines on the gradient of a scalar field, displayed simultaneously with an isosurface of the scalar field. 33

4.2 The output of a simple dataflow network, showing streamlines on the gradient of a scalar field, displayed simultaneously with an isosurface of the scalar field. 34

4.3 A portion user interface for the SolveMatrix module. The user can change the target residual by moving the small diamond on the graph. This is an example of a direct lightweight parameter change. 40

4.4 Demonstration of steering through a feedback loop in the dataflow network. Data flows beginning with an initial Mesh (generated in InsertDelaunay), performs a computation (BuildFEMatrix and SolveMatrix), refines the mesh according to an error estimate (MeshRefiner), and continues back to MeshIterator. The MeshIterator module will continue iterating the loop until the MeshRefiner module declares that no more adaptation is necessary. The final mesh is rendered using the MeshRender module. 41

5.1 Example of flexibility achieved from the SCIRun object-oriented data model. The module shown in the figure uses an iterative method to solve a matrix/vector system ($A\mathbf{x} = \mathbf{b}$). Because all matrix types (dense, sparse, etc.) are derived from the same basic type supported by the iterative solver, only one such solver is required. 44

5.2 VectorField class hierarchy showing the how the unstructured grid and regular grid versions are derived from a common base class. The dotted line to the Mesh shows that the Unstructured Grid version contains a 3D tetrahedral mesh. In addition, the field contains a Vector value for each node or element in the mesh. 47

6.1 SCIRun library organization. 53

6.2 A portion of the statistics of the custom allocator, showing bytes allocated and freed, high water marks, and spinlock statistics. Some of these statistics are also displayed for each bin. To the right of each bin a small graph shows the objects in the freelist and the objects in use for each range of sizes. 57

7.1 The user interface for the SolveMatrix module, showing the iterative methods available, the graph of the convergence and the target residual, which may be changed while the computation is in progress. 67

7.2 Visualization of the sparse structure of a matrix. The black dots represent the nonzeros in a symmetric matrix with approximately 10,000 rows and columns. 68

7.3 Visualization of a defibrillator design simulation, showing an electrode, the surface of the heart (epicardium), a 3D widget (rake), and electrical current lines (streamlines). The other electrode in the simulation is obscured by the heart. Much of the current leaving the visible electrode travels away from the heart, due to the high conductivity of blood in a nearby vessel. 70

8.1 An example of a fairly complex dataflow network, showing the modules (the boxes), the connections (the wires between them), and the input/output ports (the points on the modules that the wires connect). On a color monitor, the colors of the ports and connections indicate the type of data that flows through them. 83

8.2 Visualization of a defibrillator design simulation, showing an electrode, the surface of the heart (epicardium), a 3D widget (circular rake), and electrical current lines (streamlines). The other electrode in the simulation is obscured by the heart. Much of the current leaving the visible electrode travels away from the heart, due to the high conductivity of blood in a nearby vessel. 86

8.3 The dataflow program for a path tracing program. The imaging pipeline can be manipulated even while the program executes. 88

8.4 The Cornell Box as generated by the SCIRun PathTracer Module and imaging pipeline 89

8.5 An example of the SCIRun path tracer, showing the interface to steer the adaptation parameters while the program is still executing. 90

8.6 The dataflow program for an example Fortran program (CFDLIB). The CFDLIB module contains the entire Fortran simulation, and various visualization and postprocess modules are connected to it. 92

8.7 The dataflow program and resulting visualization for an example Fortran program (CFDLIB). The simulation consists of three fluids of different density that interact inside of a box. 93

8.8 This picture shows one component of the plume in greater detail in three perpendicular cross sections. 103

8.9 A SCIRun network for an example atmospheric diffusion simulation. 104

ACKNOWLEDGEMENTS

I thank my parents for the support and assistance through all of my education. Thanks especially to MeriAnn for her love, support and help during the completion of this degree. I also thank the members of the Scientific Computing and Imaging research group for using SCIRun, for providing great feedback, and for helping in SCIRun's construction. Work on SCIRun was made possible, in part, by grants from The National Science Foundation, the National Institutes of Health, and the Department of Energy. During much of my tenure as a graduate student, I received support in the form of a fellowship from the Department of Energy Computational Science Graduate Fellowship program. Finally, I wish to thank the members of my committee for their valuable input and encouragement.

CHAPTER 1

INTRODUCTION

Computational steering is the interactive control of scientific computation. This dissertation demonstrates that computational steering can be a useful tool in computational science, engineering, and medicine applications and that this utility is obtained by providing a flexible and efficient infrastructure whereby disparate computational tools can be used in a single, focused environment. Furthermore, we demonstrate that using such an environment, a scientist or engineer can rapidly investigate the solution space for iterative computational design problems. Computational steering can bring a degree of interactivity to these long-running, traditionally batch-oriented, scientific computations.

The system presented will demonstrate the utility of a programming environment that combines computational steering with an efficient, modular, extensible, visual programming system. These issues are investigated in the context of an implementation of such an environment, called SCIRun.[1] The primary goal of SCIRun is to provide an efficient environment that enables scientists to create new simulations and scenarios, to develop new algorithms, and to couple existing algorithms with powerful visualization tools. SCIRun combines several different computer science concepts to achieve this flexibility and efficiency. A dataflow [27] visual programming model is combined with concepts from object-oriented programming to achieve modularity by insulating data representations from dataflow modules. This combination also provides a mechanism for implementing demand driven dataflow (lazy evaluation) [120] in a straightforward manner. Threads [111] are used to provide task parallelism expressed naturally in the dataflow graph, as well as data parallelism coded explicitly in different components.

[1] SCIRun is pronounced “ski-run” and derives its name from the Scientific Computing and Imaging (SCI) research group which is pronounced “ski” as in “Ski Utah.”

Finally, we extend the concept of dataflow's internal communication by allowing more flexible communication ports. This facilitates a more expressive set of communication methods for implementing relationships that do not fit naturally in the dataflow model. These communication ports also provide a mechanism for using fine-grained dataflow mixed with coarse-grained dataflow in a flexible manner. Each of these features is examined in the context of steerable scientific simulations.

1.1 Motivation

Since the introduction of computers, scientists and engineers have used them to numerically simulate complex physical phenomena [9]. Now the computer is an almost universal tool used in a wide range of scientific and engineering domains.

Scientific and engineering disciplines are turning to scientific simulation to provide answers to questions that can not be answered through experimental or theoretical means. These simulations have three main characteristics:

- **Large-scale**. Whereas large-scale is a function of the machine availability, to get the most accurate answers possible, most scientists attempt to use as much resolution as they have memory and patience for. Large-scale means large memory requirements and/or large CPU time requirements. Although the definition of large-scale varies based on technology constraints and budget realities, we include large-scale to mean data that consume a significant fraction of machine memory, currently multiple gigabytes on high-end architectures. Similarly, the definition of long-running dependent on the speed of available machines and the patience of the user. For this work, we have addressed long-running simulations on the order of minutes to hours.

- **Iterative**. The simulations are typically performed to optimize some set of parameters or to evaluate some set of scenarios often characterized by sets of boundary and initial conditions. In these cases, the scientist often performs a simulation, makes changes in the geometry, boundary/initial conditions, solution and/or other parameters, examines the results, and then performs new simulations. Simulations may be performed using a known method, but often the scientist is developing new

solutions techniques for a particular problem and wishes to connect the changes made in the parameter space to the results of the simulation, i.e., conducting a series "what if" computational experiments.

- **Multistep**. Simulations typically are composed of a fairly complex set of steps. These steps can be categorized as modeling, simulation, and visualization steps. Data from one step are used in later steps. Traditional methods have evolved with these steps as multiple distinct programs with their own interface, and with their own unique representations of the data.

In a typical computational field problem scenario, a scientist would use the following process:

1. **Construct a model of the physical problem domain.** Modeling includes specifying the shape of the problem domain and can include specifying physical properties, such as electrical conductivity, density or viscosity. Simple problems may have relatively simple models, such as cubes, spheres, or other simple geometries. However current trends typically require the use of models that accurately portray a related physical problem domain. For example, a computational medicine problem described in Chapter 8 involves creating a detailed model of the human anatomy that describes the geometry and material properties for the bones, muscles, and organs in a human thorax.

2. **Apply boundary conditions and/or initial conditions.** Boundary conditions are the forces that drive a particular problem. Typical boundary conditions may include the velocity of wind at the input of a wind tunnel, the electrical sources for an electrical conduction problem, or boundary temperatures for a heat conduction problem. Boundary conditions are equations defined on a boundary that couple with the governing equations to define the behavior of the system at these boundaries. Initial conditions include the specification of starting data for the simulation – such as the current weather conditions for a weather simulation. Parameters associated with these equations may also be specified in conjunction with other model parameters.

3. **Develop a numerical approximation to the governing equations.** Governing equations are often a set of ordinary or partial differential equations that define the behavior of the problem. Since the computer cannot operate on these equations directly, the equations are discretized using methods such as finite difference, finite element, finite volume, or boundary element methods.

4. **Compute.** Once the data have been specified, the computer is used to solve this numerical approximation. This typically involves solving a linear or nonlinear system of algebraic equations. For realistic models, these systems of equations can be extremely large – with thousands to billions of unknowns and thus can necessitate the use of parallel computing techniques.

5. **Validate the results.** Once the solution has been found, the scientist must determine if the results are correct. Validation methods include computing known problem invariants (a form of "checksum"), comparing with experimental data, comparing with simple problems that have an analytical solution, demonstrating observed physical phenomena from "first principles," or determining that the answer is plausible according to the scientists' expertise on the problem.

6. **Understand the results.** Early computational scientists printed out stacks of numbers on continuous sheet line printers and stared at them for hours. As computers have grown more powerful, scientists have been able to perform more and more complex simulations, and quickly outgrew this form of analysis. Fortunately, these more powerful computers are also able to assist in presenting the information in a more meaningful way — forming the entire discipline of Scientific Visualization [37, 86, 102, 107]. After all, as Richard Hamming wrote, "The purpose of computing is insight, not numbers" [42, page vi]. Visualization is the science and art of turning numbers into pictures that assist in the understanding of those numbers.

Figure 1.1 exemplifies the computational process applied to modeling the electrical currents in the human torso during a normal heartbeat [56, 57, 77, 78, 106]. In this example, the modeling, simulation, and visualization process took several months and was

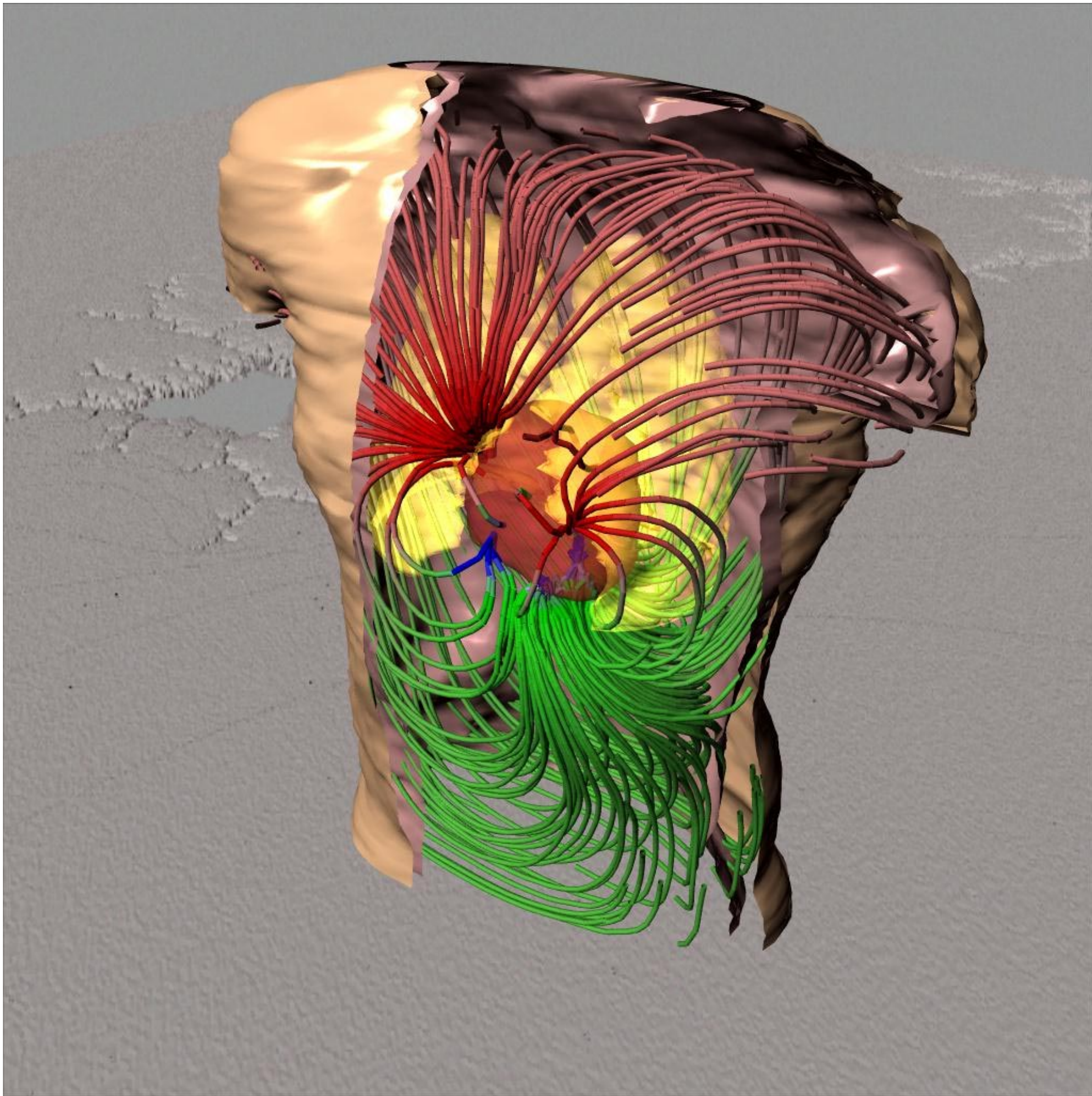

Figure 1.1. Example of the results of a computational process. This is a visualization of the electrical currents in the human torso, at one point in a normal heartbeat [56, 57, 77, 78, 106].

performed by a wide variety of different custom and commercial software packages [76].

Over the years, scientific computing has grown into a widely accepted method of scientific investigation. Scientists are continuously trying to perform more accurate simulations, to use more realistic physical models, and to obtain answers in shorter time with less work. Many scientists are also applying these techniques to new problem domains and are using them to solve new practical problems.

1.2 Computational Steering

Currently, organizing, running, and visualizing a new large-scale simulation still requires hours or days of a researcher's time. Time and effort required for data input, output and conversion further complicates and slows the process. Even for experienced scientists who may employ scripts and conversion programs to aid them in the modeling task, the process is anything but streamlined. As noted at the NSF-sponsored workshop on Health Care and High Performance Computing,[2] scientists and engineers want a system in which all these computational components are linked – they wish to "close the loop," so that all aspects of the modeling and simulation process can be controlled graphically within the context of a single application program.

In 1987, the Visualization in Scientific Computing (ViSC) workshop reported [32, page 5]:

> Scientists not only want to analyze data that results from super-computations; they also want to interpret what is happening to the data during super-computations. Researchers want to *steer* calculations in close-to-real-time; they want to be able to change parameters, resolution or representation, and see the effects. They want to drive the scientific discovery process; they want to *interact* with their data.
>
> The most common mode of visualization today at national supercomputer centers is batch. *Batch processing* defines a sequential process: compute, generate images and plots, and then record on paper, videotape or film.
>
> *Interactive visual computing* is a process whereby scientists communicate with data by manipulating its visual representation during processing. The more sophisticated process of *navigation* allows scientists to *steer*, or dynamically modify computations while they are occurring. These processes are invaluable tools for scientific discovery.

Although these thoughts were expressed 10 years ago, they express a very simple idea that scientists want more interaction than is currently present in most simulation codes. Computational steering has been defined as "the capacity to control the execution of long-running, resource-intensive programs" [40]. We refine this definition to include interactive control of all aspects of these programs. As such, steering provides a mechanism to iteratively evaluate parameter changes in a large-scale computational

[2]Held in Washington, D.C. Dec. 8-9, 1994.

environment. Furthermore, we restrict the focus of this research to computational science programs. Other long-running, resource-intensive programs such as database servers, web servers, and operating systems will not be examined here.

In the field of computational science, we apply this concept to link visualization with computation and geometric design to interactively explore (steer) a simulation in time and/or space. As knowledge is gained, a scientist can change the input conditions and/or other parameters of the simulation. Implementation of a computational steering framework requires a successful integration of the many aspects of scientific computing, including geometric modeling, numerical analysis, and scientific visualization, all of which need to be effectively coordinated within an efficient computing environment (which, for large-scale problems, means dealing with the subtleties of various high-performance architectures).

The computational steering approach is readily applicable to *computational field problems,* which encompass a large subset of science and engineering problems, including computational fluid dynamics (CFD), electromagnetic field simulation, and weather modeling – essentially any problem whose physics can be modeled effectively by ordinary or partial differential equations.

Although this particular project was initially motivated by biomedical design problems, there are many other engineering design problems that would benefit from such a system, including traditional mechanical, and CFD aerodynamics design. These engineers desire a tool with which they can easily experiment with new geometric configurations, and can also modify computational models and numerical algorithms to achieve more accurate results, or to compute the solution more efficiently. With an integrated visualization environment combined with large-scale computational capabilities, scientists would wield a powerful tool for engineering design.

1.3 Problem Solving Environments

Although steering was proposed over a decade ago, it is only gradually becoming a popular paradigm for scientific computing [15, 16, 50, 65, 123]. Computational steering is difficult because it requires in-depth knowledge in a wide range of disciplines from

geometric modeling to scientific computing to scientific visualization and graphics. Most scientists do not have the necessary expertise in visualization, and most visualization experts do not perform large-scale scientific simulations. To successfully apply computational steering to these iterative design problems, we implement a *Problem Solving Environment (PSE)* wherein these various phases of the scientific computing process may be integrated.

The environment must allow the scientist to focus on the science and the visualization expert to focus on understanding the data. As a result, a highly modular visual programming environment has been implemented that allows these disparate pieces to exist in an integrated environment.

1.3.1 Responsibilities of a Problem Solving Environment

A problem solving environment should be an efficient tool for solving all aspects of a problem. It should provide flexible modeling, visualization and computational components, but due to the continual development of new approaches, it will probably never provide a comprehensive set. As a result, it should also provide facilities for implementing, debugging, and tuning new components. The scientist should be able to use these same tools throughout the process – during development, debugging, tuning, production and publication. To provide an efficient environment for developing and controlling scientific computations, the problem solving environment assumes several responsibilities.

The first responsibility is to provide a flexible interface for reusing various modeling, computational and visualization components. The problem solving environment we have created, SCIRun, accomplishes this through a visual programming interface that allows a scientist to compose appropriate tools for analyzing and visualizing various data (including end results, intermediate results, and even debugging data).

A problem solving environment should also be responsible for providing an appropriate development environment for PSE components. Development includes initial program construction, debugging and performance tuning. Traditional debuggers are typically not efficient at dealing with the amount of data that scientific programs produce, so SCIRun allows the scientist to use the same visual analysis tools to examine and probe

intermediate results. SCIRun also provides visualizations of memory usage, module CPU usage reports, and execution states. The development environment is further enhanced through cooperation with a traditional debugger, which allows the user to closely examine internal data structures when a module fails. SCIRun employs dynamic shared libraries to allow the user to recompile only a specific module without the expense of a complete relink. Another SCIRun window contains an interactive prompt that gives the user access to a shell that can be used to interactively query and change parameters in the simulation.

Another responsibility of a problem solving environment is to ensure the efficient use of system resources. In a sophisticated simulation, each of the individual components (modeling, mesh generation, nonlinear/linear solvers, visualization, etc.) typically consume a large amount of memory and CPU resources. When all of these pieces are connected into a single program, the potential computational load is enormous. To use the resources effectively, SCIRun adopts a role similar to an operating system in managing these resources. SCIRun manages scheduling and prioritization of threads, mapping of threads to processors, interthread communication, thread stack growth, memory allocation policies, and memory access exception signals.

Steering tools and environments, such as Magellan [126] and Pablo [97], that focus on performance steering and algorithm refinement, address some of these issues. They provide mechanisms for performance tuning that can either be controlled by the user/developer or automated based upon performance statistics. However, they do not provide a rich set of components for computational steering of an application. By having an integrated steering environment for both developing and running an application, the user/developer has the capability to easily migrate from development to production. Furthermore, steering modifications that affect the performance can be more easily understood if discovered in an interactive setting.

1.3.2 Requirements of the Application

A problem solving environment provides a framework for constructing and executing steerable scientific and engineering applications. However, the application programmer must assume the responsibility of breaking up an application into suitable components.

In practice, this modularization is already present inside most scientific programs, since modular programming has been preached by software engineers as a sensible programming style for years [33].

More importantly, it is the responsibility of the application programmer to ensure that parameter changes make sense with regard to the underlying physics of the problem. In a CFD simulation, for example, it is not physically possible for a boundary to move within a single timestep without a dramatic impact on the flow. The application programmer may be better off allowing the user to apply forces to a boundary that would move the boundary in a physically coherent manner. Alternatively, the user could be warned that moving a boundary in a nonphysical manner would induce gross errors in the transient solution. SCIRun does not enforce these realities.

SCIRun does provide mechanisms to control when particular parameters get changed, so that these realities may be controlled by the programmer.

1.3.3 Implementing a PSE

As described above, the desire to interact with a running simulation has been expressed often. However, the methods for implementing the mechanisms for interaction differ tremendously, as we illustrate in the Chapter 2. In many systems, steering mechanisms limit steering to either modifications during the algorithm development phase or during the modeling and computational cycle. Problem solving environments extend capabilities by allowing similar steering mechanisms to be exploited during all phases of development, application, and performance tuning. They also allow the same visualization and analysis tools to be used during all phases.

A problem solving environment attempts to integrate a domain-specific library with a high-level user interface, consisting of a high-level language and a graphical interface, through the use of software infrastructure. Problem solving in scientific computation typically involves symbolic computation, numeric computation, and visualization. Thus, many PSEs, such as MatLab [1] from The Math Works Inc., Mathematica [2] from Wolfram Research, Maple [3] from Waterloo Maple Inc., and ELLPACK [100] from Purdue University integrate numerical libraries with visualization postprocessing. An extensive list of PSEs can be found on-line [95].

An integrated problem solving environment provides a complete set of tools for a scientist to solve a class of problems. In this context, computational steering can be a versatile tool for making changes in models, for developing new algorithms, for visualizing and analyzing results, and for tuning the performance of an application. Programming tools may be a necessary evil of the process, but the intent is for the PSE to help the scientist accurately solve a problem in a minimum amount of time. Nonetheless, scientists typically expend significant energy on programming, and they want answers to "what-if?" questions for reasoning about program bugs and for testing and improving the performance of an application.

1.4 Thesis

Computational steering can successfully unite the modeling, simulation and visualization aspects of scientific computation. A general framework can be successfully designed that allows a scientist to efficiently link these traditionally disparate phases. Through this combination, a scientist or engineer can rapidly investigate the solution space for iterative computational design problems.

This dissertation investigates these issues in the context of SCIRun [59, 60, 89, 90, 92], a problem solving environment that applies the concept of computational steering to a variety of scientific problems. SCIRun has primarily focused on computational field problems, although other applications are described as well.

Such an environment should address the following issues:

1. Interactivity: Although it is beyond the scope of this thesis to prove, many believe that interactivity plays an important role in the process of understanding [17, 45]. It is through interactivity that cause and effect relationships are revealed and is the most natural mechanism for a scientist to test hypotheses. The environment should be designed for interactivity – even for large-scale problems.

2. Integration: It should address the needs of the modeling, simulation, and visualization aspects of the problem and should allow these components to be used in chorus.

3. Extensiblity: It should not be a monolithic solution for a handful of problems but should allow the composition of various algorithms for solving new problems.

Steering a large scientific application involves much more than connecting a graphical user interface to a few parameters. Several techniques exist for extracting information from running programs, for injecting updates back into the program, and for managing these changes. We argue that these techniques are most effective when used in a highly integrated environment, where data can be shared among the various computing and visualization tasks. For this research, this integrated environment is called SCIRun.

SCIRun employs a blend of object-oriented (C++), imperative (C and Fortran), scripted (Tcl [88]) and visual (the SCIRun Dataflow interface) languages to build this interactive environment. The basic SCIRun system provides an optimized dataflow programming environment, a sophisticated C++ data model library, resource management, and development features. SCIRun modules implement components for computational, modeling, and visualization tasks.

Figure 1.2 demonstrates SCIRun, applied to a similar problem as shown in Figure 1.1. However, in this instance the scientist is free to change various parameters – mesh discretization, iterative solution method, source placements, and visualization tools displayed. The computation is steerable in an interactive investigation mode. Chapter 8 describes this application in further detail, along with a description of the application of SCIRun to a CFD simulation using a legacy Fortran CFD package (CFDLIB), the application of SCIRun to a time-dependent simulation of atmospheric diffusion from a power station plume, as well as an application of Monte Carlo based global illumination computation.

1.5 Evaluation of Thesis

Success is demonstrated in large part by a demonstration of four different applications that are implemented in SCIRun (Chapter 8). A comparison of the performance and flexibility of SCIRun and other dataflow visualization packages shows why SCIRun is more appropriate for large-scale computations. We also discuss how each of these applications extend SCIRun and how their utility has improved in a computational

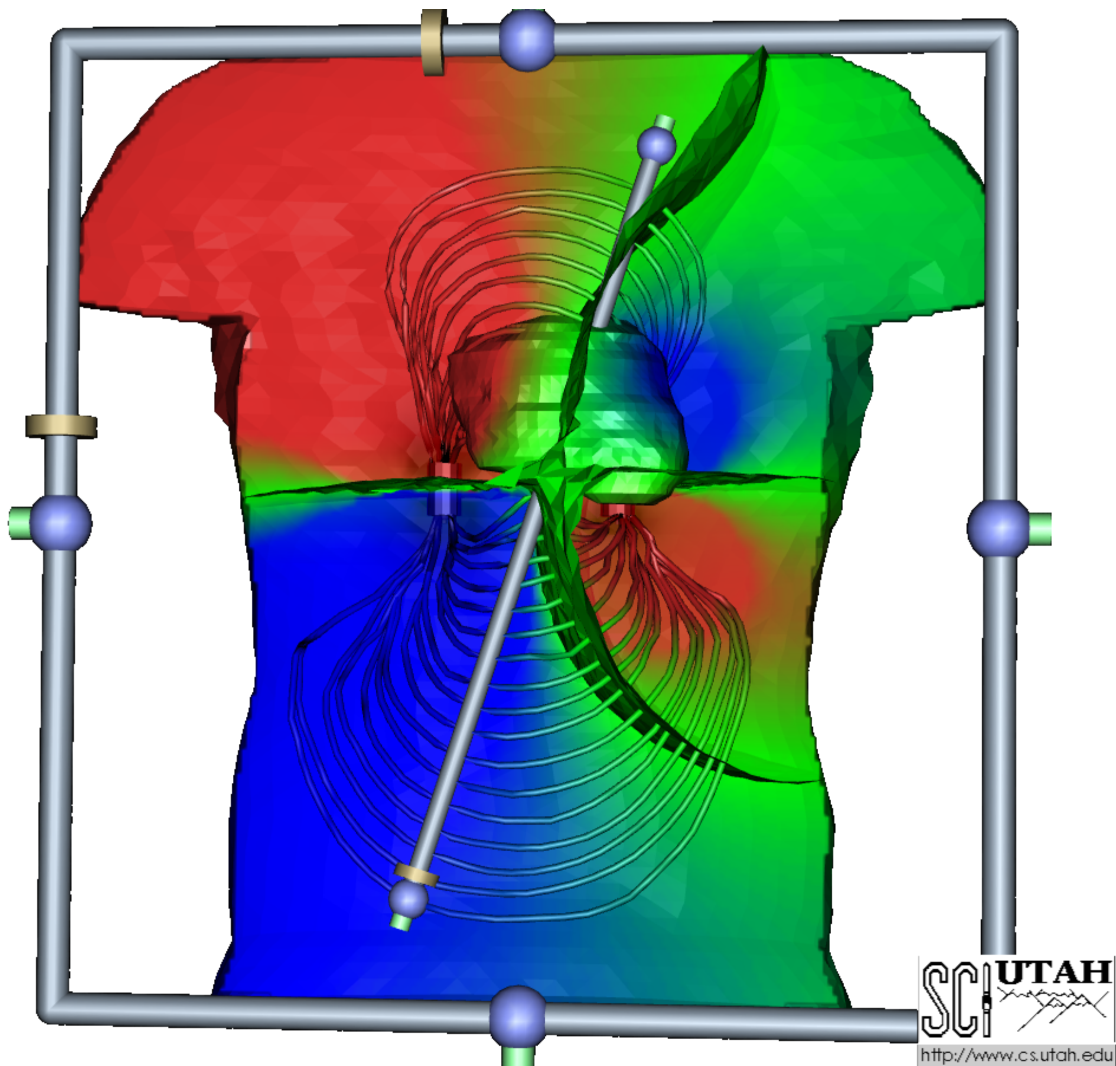

Figure 1.2. Example of an interactive computational process. Unlike the example in Figure 1.1, this visualization is steerable - the user can change the placement of the electrodes, solution methods, and other numerical and geometrical parameters.

steering environment. Each of these applications was selected to demonstrate a specific feature of SCIRun. 1) A torso defibrillator design problem demonstrates how the SCIRun components can be used to develop a new large-scale scientific simulation. 2) A Monte Carlo global illumination example demonstrates the SCIRun imaging pipeline, specifically how fine-grained dataflow can be combined with coarse-grained dataflow to restore interactivity in long-running applications. 3) A computational fluid dynamics example demonstrates how SCIRun can be used with existing code to provide steering in

a traditionally batch-oriented scientific program (CFDLIB). 4) Finally, an atmospheric diffusion example illustrates the incorporation of a time-varying adaptive unstructured grid program in the SCIRun environment. The power of the object-oriented data model is demonstrated by showing how the CFDLIB data structures can be adapted to SCIRun without explicitly duplicating and converting them. This can reduce the overall memory requirements of SCIRun and distinguishes SCIRun from other currently available dataflow systems.

1.6 Contributions

SCIRun offers a visual programming environment for scientific computing. To achieve an interactive computational steering environment, SCIRun extends the typical dataflow system by integrating concepts from object-oriented programming and by generalizing dataflow communication ports to allow for component relationships that are difficult to express using the traditional dataflow metaphor.

This work differs from traditional dataflow-oriented visual programming systems in that it focuses on efficiency for large-scale computational problems. This theme runs through many of the solutions proposed, because achieving high performance requires more than just a streamlined implementation. We demonstrate that this efficiency can be achieved while maintaining the simple composability typically associated with dataflow toolkits. Furthermore, an integrated environment can sometimes provide opportunities to gain efficiency in a manner difficult to do in the traditional environment. With a cohesive integrated environment, SCIRun components can exploit spatial and temporal coherence (explained in Chapter 3) to reduce the computational cost for iterative analysis. The first computation is performed at a fixed cost, but subsequent iterations can be faster.

Finally, we demonstrate the ability to control parameters in a running program, including legacy applications. These mechanisms provide support for controlling modelling, computational and visualization components.

1.7 Thesis Organization

In the remainder of the dissertation, we discuss related systems (Chapter 2) and then describe the SCIRun architecture (Chapter 3). We describe the dataflow implementation

and the semantics of steering in a dataflow environment (Chapter 4). A description of the SCIRun object-oriented data models (Chapter 5), the computational support libraries (Chapter 6), and the SCIRun module system (Chapter 7) shows how SCIRun modules are implemented, and the ways in which SCIRun can be extended. Chapter 7 also contains a description of several of the important modules that have been implemented in SCIRun to date. Results and success evaluation are be discussed in Chapter 8. Conclusions are discussed in Chapter 9, and future work is discussed in Chapter 10. Finally, the Appendix describes some of the design decisions made in the construction of SCIRun and opinions of how those turned out.

1.8 Publications

Parts of the research results contained in this dissertation have been published in the following journals, conference proceedings, and book chapters. This dissertation supersedes the descriptions of SCIRun in these publications, but many of them contain greater focus on particular applications and tools.

- JOHNSON, C., BERZINS, M., ZHUKOV, L., AND COFFEY, R. SCIRun: Application to atmospheric dispersion problems using unstructured meshes. In *Numerical Methods for Fluid Dynamics VI.* (1998), M. Baines, Ed.

- MILLER, M., HANSEN, C., PARKER, S., AND JOHNSON, C. Simulation steering with scirun in a distributed memory environment. In *Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)* (July 1998).

- PARKER, S., MILLER, M., HANSEN, C., AND JOHNSON, C. An integrated problem solving environment: The SCIRun computational steering system. In *31st Hawaii International Conference on System Sciences (HICSS-31)* (1998).

- PARKER, S., WEINSTEIN, D., AND JOHNSON, C. The SCIRun computational steering software system. In *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen, Eds. Birkhauser Press, 1997, pp. 1–44.

  pp. 1-44, 1997.

- PARKER, S., BEAZLEY, D., AND JOHNSON, C. Computational steering software systems and strategies. *IEEE Computational Science and Engineering 4*, 4 (1997), 50–59.

- GITLIN, C., AND JOHNSON, C. MeshView: A tool for exploring 3D unstructured tetrahedral meshes. *Proceedings of the 5th International Meshing Roundtable*, (1996), 333–345.

- PARKER, S., AND JOHNSON, C. SCIRun: Applying interactive computer graphics to scientific problems. In *SIGGRAPH '96 Visual Proceedings* (1996).

- JOHNSON, C., AND PARKER, S. Applications in computational medicine using SCIRun: A computational steering programming environment. In *Supercomputer '95* (1995), Springer-Verlag, pp. 2–19.

- PARKER, S., AND JOHNSON, C. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95* (1995), IEEE Press.

- JOHNSON, C., AND PARKER, S. A computational steering model applied to problems in medicine. In *Supercomputing '94* (1994), IEEE Press, pp. 540–549.

CHAPTER 2

RELATED WORK

Several tools and environments for computational steering have been developed. These range from tools that modify performance characteristics of running applications, either by automated means or by user interaction, to tools that modify the underlying computational application, thereby allowing application steering of the computational process. The roles of a problem solving environment should encompass all of these characteristics, from algorithm development through performance tuning to application steering, for scientific exploration and visualization. It should also provide a rich environment for accomplishing computational science.

Implementation of a computational steering environment requires a successful integration of the many aspects of scientific computing, including performance analysis, geometric modeling, numerical analysis, and scientific visualization. These requirements need to be effectively coordinated within an efficient computing environment (which, for large-scale problems, means dealing with the subtleties of various high-performance architectures).

An excellent coverage of computational steering research can be found in [40] and a more recent survey of narrowed scope can be found in [123].

2.1 Taxonomy of Steering Systems

The area of computational steering is fairly young, but there are many systems and tools that exist to assist programmers and scientific researchers in tuning and running scientific codes. It is helpful to think of these computational tools and systems within a conceptual framework to compare and contrast them [91]. In the following section we review the work of others who previously sought to classify computational steering

systems. Afterwards, we present a cohesive taxonomy for describing computational steering systems and toolsets.

2.1.1 Previous Classifications

Burnett et al. [19] propose a taxonomy for computational steering using visual languages. Visualization systems studied vary on a continuum from postprocessing through tracking to interactive visualization to steering. Interfaces presented ranged from a textual interface to a graphical user interface to a visual programming language interface. The authors argue for a merging of the interactive experimentation allowed by steering capabilities and the ease of use of a visual programming language for a researcher not trained in programming.

Vetter and Schwan [125] delineate two types of steering in existing systems: human-interactive steering and algorithmic steering. In human-interactive steering, a person monitors the computation and manipulates parameters of the computation while it is executing. In algorithmic steering, the computer makes decisions by monitoring runtime statistics and other information sources such as history files. Vetter and Schwan describe a simple feedback model for computational steering wherein output is monitored by a steering agent, either human or algorithmic. The steering agent performs steering actions (which can be changes to the parameters of the computation) based on monitored inputs. They provide examples demonstrating the steering of an application's performance (load-balancing), which automatically adapts the distributed load based upon run-time statistics.

As noted by Burnett et al. for human-interactive steering, the mechanism of interaction affects the ease of use of the system to a scientist. Systems range from providing a textual interface from which to steer to providing a graphical interface. A visual programming language could foster the creation of a steering environment that allows the user to view the program, the simulation, and the steering mechanism potentially all at the same time. On the other hand, algorithmic steering would be programmed entirely behind the scenes, but would require more programming expertise and knowledge of the computational techniques used to solve the problem.

Although both of these classifications provide insight into differing tools and applica-

tions for simulation steering, they provide orthogonal views. Burnett's work focuses on the level of steering and the visual interface whereas Vetter's classification is based upon whether the steering process can be automated.

Next, we review existing tools for simulation steering and present a different taxonomy which attempts to highlight the richness of a simulation steering environment or toolset.

2.1.2 Existing Tools for Steering

Since the idea of computational steering was first proposed, there have been several efforts at integrating steering in the computational science process. Most of these are targeted towards instrumenting existing scientific applications for use in a steering mode.

2.1.2.1 Lightweight Steering: Scripting Languages and Wrappers

Beazley and Lomdahl [15] demonstrate the use of a lightweight method of steering a large-scale molecular dynamics simulation. Using a Simplified Wrapper Interface Generator (SWIG) to wrap existing simulation codes, a scientific researcher can easily build a scripting language interface, such as Tcl/Tk [88] or Python [75], for steering a computation. Their work highlights the ease of converting existing scientific codes into a form in which they can be glued together by a control language. Then, the researcher monitors and manipulates the computation or simulation using scripting commands. Clearly, this method requires knowledge of how to program with scripting languages and does not explicitly constitute a steering toolkit. Nonetheless, steerable applications have been created using SWIG [14].

2.1.2.2 CUMULVS

The CUMULVS library [36, 66], developed at Oak Ridge National Laboratory, acts as a middle layer between Parallel Virtual Machine (PVM) applications and existing visualization packages such as AVS 5.0. After initializing a viewer, the application programmer can provide a list of parameters to be adjusted on-the-fly in a CUMULVS steering initialization procedure call. Separate procedure calls are used for altering scalar or vector parameters from within the application. CUMULVS supports multiple views

of the same running application to assist collaborations. An interesting check-pointing capability for rolling back and restarting a failed program run has the potential to allow cross-platform migration and heterogeneous restart of an application.

2.1.2.3 Progress and Magellan

The Progress Toolkit [124], developed at the Georgia Institute of Technology (GIT), assists application programmers in developing steerable applications. Programmers instrument their applications with library calls, using "steerable objects," which can be altered at runtime through the use of the Progress runtime system. Steerable objects include sensors, actuators, probes, function hooks, complex actions, and synchronization points. Progress uses a client/server program model.

Developed by the same group, the Magellan Steering System [126] is derived from the Progress system and extends the steering clients and steering servers model used in the initial system. This system uses a specialized specification language, ACSL, which provides commands for monitoring and steering using probes, sensors, and actuators. However, application codes still must be instrumented with these commands to utilize the steering capabilities of this system. These systems have been used for Molecular Dynamics simulations.

Both systems are layered on top of the Falcon system [39], also developed at GIT, which monitors a running program, capturing information ranging from a single program variable, much as a debugger would, to complex expressions. It also permits the monitoring of performance data, with interfaces to visualization systems, such as Iris Explorer [48]. Decisions about which steering actions to take are based on previously encoded routines stored in a steering event database located on a steering server.

2.1.2.4 VASE

The Visualization and Application Steering Environment (VASE) [49], from the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign (UIUC), provides a toolset for interactive visualization and steering in a distributed environment. The VASE user model identifies three distinct roles: an application developer who writes the scientific codes, a configurer who sets up the distributed envi-

ronment (including interprocess communication), and an end user (or researcher) who uses and steers the application. Steering is accomplished through the use of *steerable locations* (programmer-defined breakpoints), altering the values of variables and parameters, and adding programming statements and scripts as the computation proceeds. VASE uses a control-flow programming model, which is displayed to the end user to guide steering. VASE allows algorithm refinement through the use of script modification at run-time. Thus, the steering process can modify not only the computational parameters and performance characteristics but also the actual code.

2.1.2.5 Pablo

The Pablo performance environment [97], also designed at UIUC, provides library routines for instrumenting source code to extract performance data as the code executes. This system follows the Falcon model of utilizing sensors to collect information from the executing code, and altering system characteristics or parameters through the use of actuators. It seeks to tune the performance of running applications as they execute. Two different models for performance-directed adaptive control (or performance steering) are discussed: closed-loop adaptive control and interactive adaptive control. First, a neural network classifies file access patterns qualitatively to change the file policy on-the-fly, varying cache size and cache block replacement policy as needed by the executing code. Second, to enable human-interactive steering, Pablo developers argue for the use of an immersive environment, specifically the Avatar virtual environment prototype [98] built at the UIUC and the National Center for Supercomputing Applications.

2.1.2.6 CSE

The Computational Steering Environment (CSE) [121] provides a centralized data manager, around which computational and visualization components (called sattelites) can be implemented. The data manager provides a subscribe/notify interface to inform satellites of changes made in the data. Any satellite is capable of effecting changes to the data. An interactive graphics editing tool allows users to sketch an interface and bind variables to user interface components.

2.1.3 A Taxonomy for Steering

After examining the tools presented above, we identified three distinct types of steering in current systems: application steering, algorithm refinement, and performance steering. Application steering refers to the capability to modify the computational process through parameter changes, mesh modifications, or other changes that affect the computational aspects of the simulation. Steering by algorithm refinement allows the underlying code to be modified or refined at runtime. Performance steering focuses on changing computational resources that affect the simulation performance such as load balancing, I/O, cache strategies or other performance related parameters.

Similarly, there is a continuum of interaction strategies from textual to visual programming that provide means for a user to interactively steer the computation. It should be noted that any steering modification can also be accomplished through automated means (i.e., requiring no end user interaction), as described in some of the systems above. Figure 2.1 places these systems within this steering taxonomy. Arrows extend from each system to show a range of interaction possibilities.

SCIRun is designed to allow many forms of interaction for scientists within a standalone system. It provides application steering and algorithm refinement but currently provides little true performance steering. Scripting, although mostly limited to text-only manipulation, spans the gamut of steering functionality, permitting performance steering, algorithm refinement, and application steering. Other systems fill a steering niche, such as Pablo's focus on performance steering. Finally, some systems, such as Progress, Magellan, Falcon, and CUMULVS, provide a range of steering functionality and many forms of interaction when coupled with a visualization system such as AVS or IRIS Explorer.

2.2 Visualization Systems

In addition to the computational steering systems described above, SCIRun can be compared to several commercial visualization packages, such as AVS 5.0 [13] (from Advanced Visualization Systems), Iris Explorer [48] (from the Numerical Algorithms Group), Data Explorer [31] (from IBM), and the Visualization Toolkit [107] from Kit-

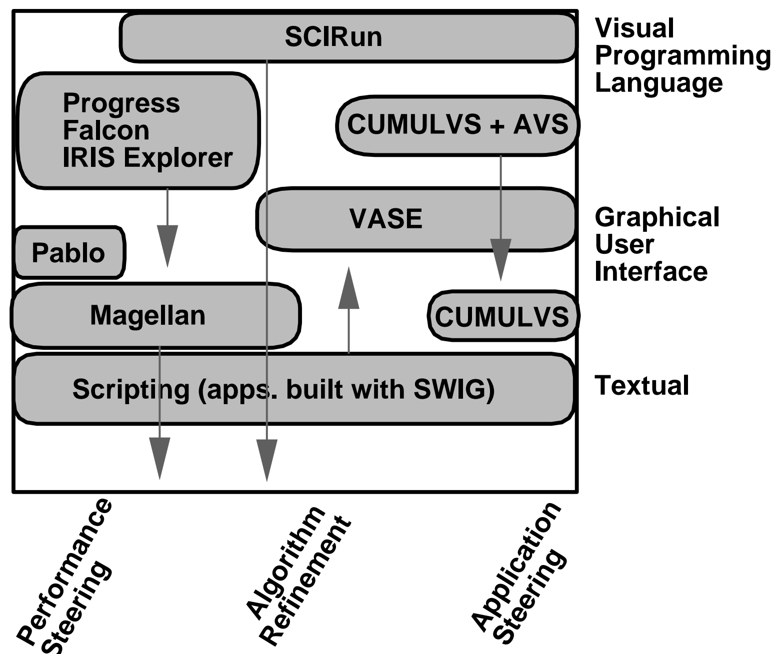

Figure 2.1. Taxonomy of steering systems and tools. The horizontal axis represents three different types of steering, and the vertical axis represents the method of interaction. The bubbles represent each of the systems' predominant strength. Arrows indicate portions of the space that are also covered by the system.

ware, Inc. These systems employ a dataflow-based visual programming environment [131]. Due to their heritage [79], these systems have primarily been used in a visualization environment where the simulation is a single module or data are read from a file. As described above, they have also been used as a visualization engine for computational steering toolkits.

The first three of these systems are compared in detail in [131]. Several important differences between these systems and SCIRun are discussed in Chapter 8.

2.2.1 AVS

Advanced Visualization Systems (AVS) was the first company to market a dataflow system for scientific visualization. It uses a multiple process model, in which each module is implemented as a separate Unix process or groups of modules are a separate process. The memory and CPU overhead for transferring large datasets from process to process can be quite large.

AVS uses a "field" datatype, which is significantly less general than the SCIRun

"Field" object. The AVS version specifies a three-dimensional (3D) structured grid, with either rectilinear or curvilinear coordinates. A different type, UCD, implements an unstructured grid.

2.2.2 Iris Explorer

Iris Explorer, originally developed at Silicon Graphics, Inc. and currently licensed by the Numerical Algorithms Group (NAG), provides a more efficient mechanism for transporting data between modules. A shared memory "arena" is used to pass data from module to module. Each module is implemented as separate Unix process.

Like AVS, Iris Explorer also provides explicit datatypes for regular (including rectilinear and curvilinear) and unstructured grids.

2.2.3 Data Explorer

IBM Data Explorer (DX) provides a single, unified data model for all types of scientific data. The model is very general, although it notably does not provide support for hierarchical grids or for higher order interpolation.

The Flow Executive in DX manages the execution of all modules. Consequently, it can become a bottleneck in the handling of data.

2.2.4 vtk

Like SCIRun, vtk uses the Tcl/Tk scripting language and user interface toolkit. However, vtk makes much more extensive use of it than SCIRun. Unlike the previous systems, vtk uses a dataflow "pull" model, which is discussed further in Chapter 4.

2.3 Chapter Summary

We have described two sets of tools: computational steering systems and visualization systems. The computational steering systems were compared in a taxonomy which describes what they were designed for, as well as the mechanisms used to interact with the simulations. Note that none of these systems have addressed a holistic solution to applying interactive computational steering in an integrated, extensible environment.

CHAPTER 3

SYSTEM OVERVIEW

The work this dissertation describes is embodied in a problem solving environment called SCIRun. SCIRun is a scientific programming environment that allows the interactive construction and steering of large-scale scientific computations. A scientific application is constructed by connecting computational elements (modules) to form a program (network). This program may contain several computational elements, as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. Geometric inputs and computational parameters may be changed interactively, and the results of these changes provide interactive feedback to the investigator.

SCIRun employs a blend of object-oriented (C++), imperative (C and Fortran), scripted (Tcl [88]), and visual (the SCIRun Dataflow interface) languages to build this interactive environment. The basic SCIRun system provides an optimized dataflow programming environment, a sophisticated C++ data model library, resource management and development features. SCIRun modules implement components for computational, modeling and visualization tasks.

3.1 Design Goals

The primary goal of SCIRun is to provide an efficient environment that enables scientists to create new simulations and scenarios, to develop new algorithms and to couple existing algorithms with powerful visualization tools.

The main design principles and goals that guided SCIRun development are as follows:

- Use visual programming and user interfaces that allow a person to utilize particular components without in-depth expertise on that particular component. This allows

scientists to use visualization tools while working on computational algorithms and a programmer to create visualization tools without implementing the simulation as well.

- Use visualization and numerical feedback throughout the system. This includes both application level visualization as well as system and algorithm level feedback.

- Rather than provide general purpose tools to solve all problems, provide a general purpose software architecture in which different scientists can use their special purpose tools to solve a particular problem.

- Provide generality wherever possible, but provide efficiency everywhere. Typically we attempt to provide generality but compromise that generality where efficiency would suffer significantly. In this context, efficiency applies to portions of the program that take a significant portion of the execution time.

- Provide support for parallelism. For the sake of simplicity, we restrict ourselves to parallelism in a multithreaded, shared memory environment.

- Provide support for modularity at a binary level. Particular components should be able to operate on data that were not known at compile time.

- Focus on 3D problems. Many have found it difficult to extend two-dimensional (2D) solutions to three dimensions. This, combined with the need to accommodate large-scale computational engineering, science, and medicine problems problems, motivated us to concentrate on 3D problems.

3.2 Applications

SCIRun was initially motivated by a set of applications in Computational Medicine [53, 54, 51, 57, 56, 58, 76, 106]. However, it is designed to be a general problem solving environment that can be used to solve a wide range of scientific and engineering problems.

In addition to the original computational medicine applications, other applications have been selected to further expand and test the generality of the system. These applications were chosen to exercise SCIRun on important scientific problems. The problems

chosen utilize a variety of data representations and computational techniques, in order to ensure that the system is sufficiently general. They are also large-scale problems (in memory and/or CPU time) that stress the efficiency of the system in general. The driving applications are as follows:

1. Torso defibrillator modeling: An implantable defibrillator is a small electrical device that is surgically implanted in the thorax and that discharges electrical shocks to defibrillate the heart when it undergoes a ventricular fibrillation. There are many variables in the design of such a device. Optimal operation is achieved only through the correct combination of voltage, placement, size, shape, and numbers of electrodes. This large array of possible design and placement combinations cannot possibly be explored experimentally (through the surgical implantation of devices in actual human subjects), but computational models can allow an engineer to test a wide range of design and placement possibilities. To accurately model such a system, a highly detailed computational model must be used, and these models can take hours of CPU time to compute the results [106, 83, 11].

2. Monte Carlo global illumination: Monte Carlo methods are used as an effective method for solving the rendering equation [61, 110]. Although the focus of this dissertation does not include rendering techniques, this application is used to demonstrate the use of SCIRun for an imaging pipeline, as an example of fine-grained dataflow, and use of the components for an application outside of computational field problems.

3. Computational fluid eynamics: CFDLIB [4] is a computational fluid dynamics package from Los Alamos National Laboratory used to compute a 3D flow. It is a large Fortran program, which epitomizes many of the qualities of a typical legacy scientific application. This application is primarily intended to demonstrate the application of computational steering to a legacy application. Special mechanisms are demonstrated that allow an application to be incrementally modified to support computational steering.

4. Atmospheric diffusion: Another application [52] is taken from a model of atmospheric dispersion from a power station plume – a concentrated source of NO_x emissions. The photo-chemical reaction of this NO_x with polluted air leads to the generation of ozone at large distances downwind from the source. An accurate description of the distribution of pollutant concentrations is needed over large spatial regions to compare with field measurement calculations.

These applications are described in further detail in Chapter 8. Many more applications are possible, with the system, some of which are described in Chapter 10 as future and current work.

3.3 SCIRun

SCIRun composes computational algorithms with these data elements using a dataflow style "boxes and wires" approach. An example of a dataflow network is shown in Figure 3.1.

1. A **module**, drawn as a box in the network, represents an algorithm or operation. A set of input ports (top) and a set of output ports (bottom) define its external parameters.

2. A **port** provides a connecting point for routing data to different stages of the computation. Ports are strongly typed: each datatype has a different color, and datatypes cannot be mixed. In SCIRun, ports can be added to and removed from a module dynamically. Input ports are represented on the top of the module icon, and output ports are on the bottom. Output ports can optionally cache datasets to avoid recomputation.

3. A **datatype** represents a concept behind the numbers. Datatypes are quantities such as scalar fields or matrices, but are not the specific representations, such as regular grids, unstructured grids, banded matrices, sparse matrices, etc.

4. A **connection** connects two modules together: the output port of one module is connected to the input port of another module. These connections control where

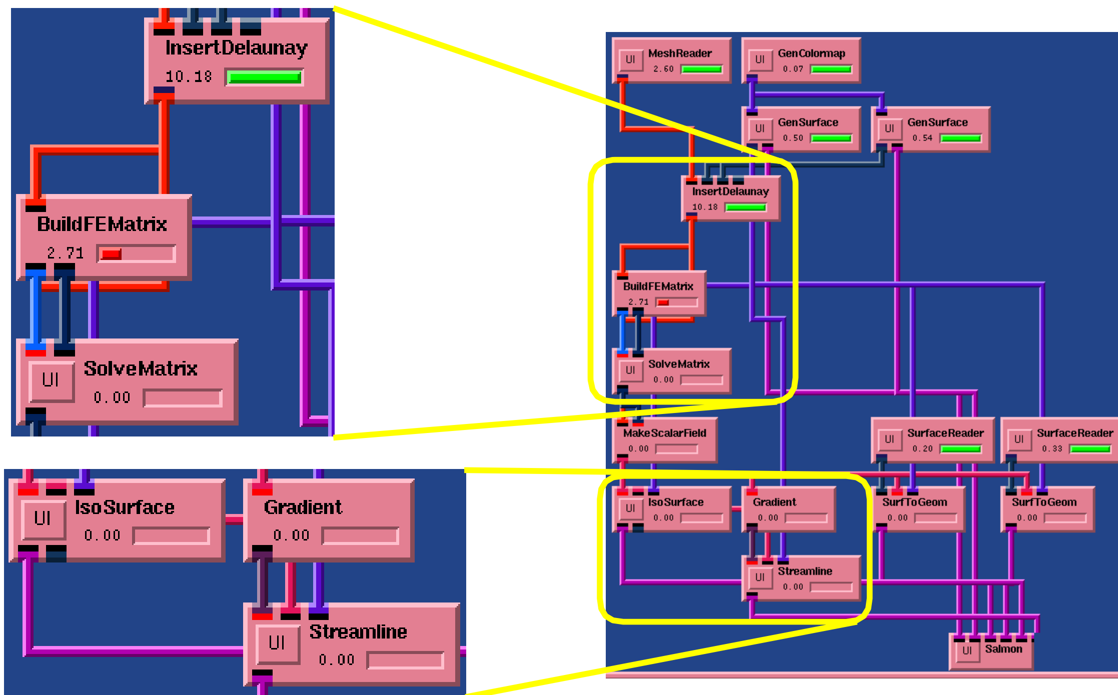

Figure 3.1. An example of a fairly complex dataflow network, showing the SCIRun modules (the boxes), the connections (the wires between them), and the input/output ports (the points on the modules that the wires connect). Magnified portions show simulation and modeling components (top left) connected to visualization components (bottom left) in a cohesive environment.

the data are sent to be computed. Output ports can be connected to multiple input ports, but input ports accept only a single connection. A module that should accept an arbitrary number of inputs can use a callback mechanism to create a new empty port when the other input ports are full.

5. A **network** consists of a set of modules and the connections between them. This represents a complete dataflow "program."

3.4 Steering Optimizations

To accommodate the large datasets required by high resolution computational models, we have optimized and streamlined the dataflow implementation. These optimizations are made necessary by the limitations many scientists have experienced with currently available dataflow visualization systems [99].

3.4.1 Data Structure Management

Many implementations of the dataflow paradigm use the port/connection structure to make copies of the data. Consider the example in Figure 3.2. If the vector field is copied to both the Hedgehog and Streamline modules, then twice as much memory is consumed as necessary. In addition, a significant amount of CPU time is required to copy large, complex data structures. To avoid these overheads, we employ a simple reference counting scheme with *smart pointers* [116] in C++. This scheme helps reduce complexity by allowing different modules to share common data structures with copy-on-write semantics. When the Gradient module creates the VectorField, it sets a reference count in the vector field to zero. As Gradient hands a copy of the vector field data structure to each of the downstream modules, the receiving module increments the reference count. When each module is finished with the data structure, it decrements the reference count. When the reference count reaches zero, the object is destroyed. These reference counts are maintained automatically by C++ classes (the smart pointers) to reduce programmer error.

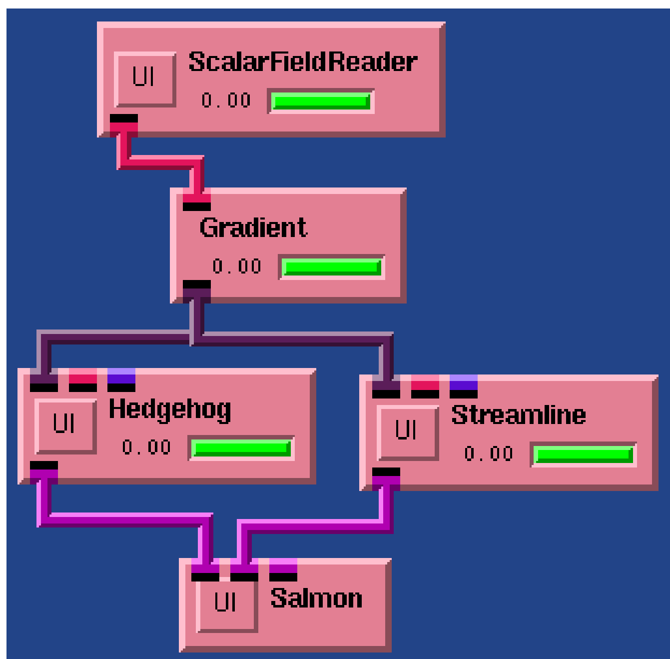

Figure 3.2. A closeup view of a dataflow network. A vector field, produced by the Gradient module, is consumed by both the Streamline and Hedgehog modules. In SCIRun, the data are shared between the modules so that the data do not need to be duplicated.

Copying the object is necessary only when a module needs to change a data structure and the reference count is greater than one (i.e., another module is also using it).

3.4.2 Progressive Refinement

Due to memory and speed limitations of current computing technologies, it will not always be possible to complete these large-scale computations at an interactive rate. To maintain some degree of dynamic interactivity, the system displays intermediate results as soon as they are available. Such results include partially converged iterative matrix solutions, partially adapted finite element grids, and incomplete streamlines or isosurfaces. In the defibrillator design example shown above, the user moves an electrode and sees some feedback almost immediately. The solution continues to converge to the final solution. In this way, an engineer or scientist can watch a solution converge and, based on the results observed, may either decide to make changes and start over or allow the simulation to continue to full convergence.

3.4.3 Exploiting Interaction Coherence

A common interactive change consists of moving and orienting portions of the geometry. Because of the nature of this interaction, surface movement is apt to be restricted to a small region of the domain. Using information available from both how the geometry has moved and its position prior to the move, the system can anticipate results and "jump start" many of the iterative methods [59]. For example, iterative matrix solvers can be jump-started by interpolating the solution from the old geometry onto the new mesh. When changes to the model geometry are small, the resulting initial guess is close to the desired solution so the solver converges rapidly. This concept is similar to exploiting temporal coherence in a time-dependent system by using the previous time-step as the initial guess to the next time step. An even more compelling example is seen in the mesh generation process for the torso defibrillator modeling problem. Typically, mesh generation for the entire torso model would take tens of minutes to hours. Since we know that the user only wants to move the defibrillator electrodes, we generate the mesh without the electrodes beforehand. Then, when the user selects an electrode placement, nodes for the defibrillator electrodes are placed into the mesh in only a few seconds.

For most boundary value and initial value problems, the final answers will be the same for the incremental and brute-force approaches (subject to numerical tolerances). However, for nonlinear problems where there may be multiple solutions or for unsteady (parabolic) problems, results may be completely different. In these instances, the interaction coherence should not be exploited or results will not be scientifically repeatable.

Through coupling each of these techniques, we are able to introduce some degree of interactivity into a process that formerly took hours, days or even weeks. Although some of these techniques (such as displaying intermediate results) add to the computation time of the process, we attempt to compensate by providing optimizations (such as exploiting interaction coherence) that are not available with the old "data file" paradigm.

3.5 Chapter Summary

We presented an overview of the SCIRun system, including the design goals that were used to guide development. Four applications were introduced: Torso defibrillator modeling, Monte Carlo global illumination, a CFD application using CFDLIB, and a simulation of atmospheric diffusion. These applications were selected to explore the gamut of SCIRun's possibilities. We presented three optimizations that are used in SCIRun to gain efficiency when performing large-scale simulations. Data structure management helps SCIRun operate on large-scale datasets without the memory overhead typically associated with these tools. Progressive refinement facilitates a degree of interactivity on long-running simulations that are not interactive. Finally, interaction coherence can actually provide an improvement in the speed of repetitive simulations through exploiting spatial and temporal coherence. These are issues that have not been addressed in current computational steering or visualization systems.

CHAPTER 4

DATAFLOW IMPLEMENTATION

In this chapter we present an overview of the SCIRun software architecture. SCIRun composes computational algorithms (Modules) with data elements (Datatypes) using a "boxes and wires," dataflow style approach [64]. An example of a simple dataflow network is shown in Figure 4.1, and the output of this network is shown in Figure 4.2.

This chapter will explain how demand-driven and data-driven dataflow mechanisms can be mixed to provide flexibility in creating modules. We also demonstrate how the communication mechanisms between modules can work to provide components that are both steerable and efficient. Finally, we will demonstrate how computational steering

Figure 4.1. An example of a simple dataflow network, used to compute streamlines on the gradient of a scalar field, displayed simultaneously with an isosurface of the scalar field.

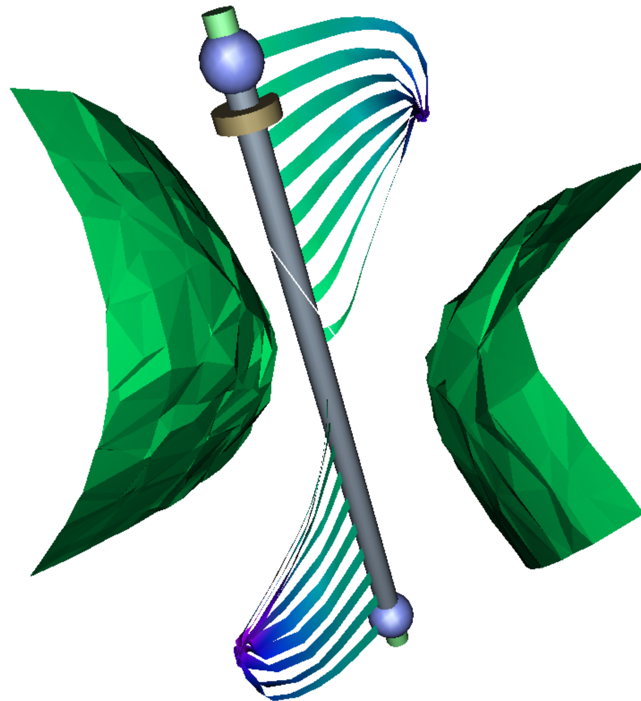

Figure 4.2. The output of a simple dataflow network, showing streamlines on the gradient of a scalar field, displayed simultaneously with an isosurface of the scalar field.

can be used in a dataflow system.

The dataflow library is responsible for deciding which modules need to be executed and when. For this discussion, "execution" consists of a single module running from beginning to end. The module typically reads data from the inputs ports (if any), performs some operation, and sends the results through one or more output ports. A module is executed again when the user changes a module parameter, when new data is available at an input port, or when data is required at its output port.

When the user moves a user interface component, such as a 2D slider or a 3D widget [96], the module sends a message to the scheduler requesting execution. The scheduler analyzes the dataflow graph to determine what other modules also need to be executed. The dataflow scheduler uses the following algorithm to determine which modules need to be executed:

```
execute_list = modules that requested execution;
resend_list = empty;
foreach module in the execute_list {
```

```
  foreach module connected to an output {
    if the connected module is not in the execute list,
      add it
  }
  foreach module connected to an input {
    if the connected output port has a cached dataset,
      add it to the resend list
    else add it to the execute_list
  }
}
cull all ports from the resend_list whose modules appear
   in the execute list
send resend messages to the ports in the resend list
send execute messages to the modules in the execute list
```

The algorithm traverses the dataflow graph, beginning at the modules that request execution. It traverses upstream (from input to output ports) until it finds a module that has cached data on the output port. Those output ports are added to a list of data that must simply be resent. Modules that do not have cached data are added to the list of modules that must be executed, and the algorithm continues upstream until a cached dataset is found or a module without input ports is found. All modules downstream of executed modules are also added to the execute list. Finally, messages are sent to the modules that must either execute or resend data. The modules receive this message from a First-in/First-out (FIFO) queue and perform the requested action.

Note that execute messages are sent to all of the modules in the execute list at the same time. The modules subsequently communicate directly with each other using a thread-safe FIFO for the dataset hand-offs. Each thread waits for data to appear on the input ports. A module is not required to gather data from the inputs in order, and it may interleave computation with receiving data. However, to satisfy the requirements for determinacy in a dataflow program, the module is not able to request datasets in a first-come, first-served or any other nondeterministic order [27, 63]. It is important to note that modules do not send actual data, but rather a handle to the data (see 6.3.2 for a discussion of handles).

Sending a dataset to the `matrix_out` port is just a few lines of code, similar to:

```
MatrixHandle matrix = new DenseMatrix(nrows, ncols);
.. build the matrix ..
matrix_out->send(matrix);
```

and receiving a dataset from the `matrix_in` port is similarly minimal:

```
MatrixHandle matrix;
if(!matrix_in->receive(matrix))
  return; // returns false if the dataset is not
          // available - this is usually due to
          // an error upstream

.. use the matrix ..
```

During the course of a single execution, the module must perform exactly one receive for each input port, and for each output port, the module must perform exactly one send. If the module wishes to send multiple datasets (as intermediate results or for feedback loops), it can use a special method called `send_multi`, which also arranges for the modules downstream to be executed again.

We use a centralized scheduler to avoid redundant module reexecution in a branching situation. Since we leave the central scheduler out of the loop for individual dataset handoffs, it does not become a bottleneck in the execution of the program.

The Dataflow library also contains a base class, `Module` from which all modules are derived. This class contains the data structures that are required to implement the dataflow structures; it also contains various utility functions, such as `update_progress`, a function that the module writer can call periodically to update the graph on the module icon that indicates the approximate percentage of work the module has completed.

4.1 Demand-Driven Dataflow

The dataflow scheduling algorithm implements a "data-driven" dataflow approach. Another approach is to use a "demand-driven" or lazy-evaluation scheme. According to Treleaven et al. [120, page 93];

> Basically, in data-driven (e.g., data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g., reduction) computers the requirement for a result triggers the operation that will generate it.

Although this statement was made in connection with computer hardware architectures, they are equally applicable to software dataflow implementations.

Most dataflow systems, including AVS 5.0 [13] (from Advanced Visualization Systems), Iris Explorer [48] (from the Numerical Algorithms Group) and Data Explorer [31] (from IBM), implement an entirely data-driven approach. The Visualization Toolkit [107] from Kitware, Inc. implements an entirely demand-driven approach.

It is difficult to know which of these approaches is best. Consider a scalar field dataset that undergoes a series of computationally intensive processing steps. The processed dataset is then visualized by a variety of visualization algorithms. In a data-driven system, the processing steps will all occur on the entire dataset before the visualization tools execute. In a demand-driven system, only the portions of the dataset that are actually examined by the visualization tools will be computed as demanded by the visualization tools. However, unless complicated caching schemes are implemented, many portions of the dataset may be recomputed multiple times as the downstream visualization modules use the data. For example, a volume rendering application may examine each cell multiple times while rendering the final image. In this case, a data-driven approach would likely be more efficient since the system would perform all computation up front, and the computation would only be performed once. However, if a different visualization tool is chosen, such as a cutting plane, the opposite would be true. A data-driven approach would spend time computing portions of the dataset that will never be seen. When using a combination of visualization tools, the choice becomes especially unclear.

To achieve the benefits of both worlds, SCIRun uses a data-driven approach in the dataflow graph, but allows a demand-driven method to be implemented using the object-oriented data model. In the example above, the processing modules can simply pass a functor [116], or an object that will perform the requested computation on demand. The downstream modules would then access the functor as a dataset, which would compute the data on demand. The object-oriented data model will be described in further detail in Chapter 5.

This approach affords a simple, efficient data-driven dataflow implementation that can also be used where a demand-driven approach would be more efficient. However, it is difficult to know a priori when one approach would be more efficient than the other. Therefore, we leave the choice to algorithm implementor and recommend that if a clear

choice is not possible that the choice be controlled by the user, with a reasonable default.

One weakness to this approach is that the modules that require both a demand-driven and data-driven implementation will need to implement the same algorithm in two different ways. Although it is extra work, it is beneficial in many cases; the algorithm would be implemented differently for operating on a dataset in bulk than for operating on a point by point basis.

4.2 Flexible Dataflow Ports

The dataflow programming paradigm naturally captures the concept of a producer-consumer relationship. In many modern simulations, other relationships are often used which more naturally fit with the design of the system. In SCIRun, the Ports that are used to connect one module to another have been relaxed from the traditional dataflow model.

The most common implementation of a Port was shown above in the Matrix example. However, each type of data may have its own definition of a port that has other communication protocols than the simple send/receive methods shown in that example.

4.2.1 Non-dataflow Ports

The Salmon module is a 3D graphical viewer application. It is described in further detail in Chapter 7. Salmon collects geometrical objects from a variety of visualization modules and displays them in a common window. The clients of Salmon (visualization modules) need to be able to send objects to Salmon. However, we would like to relax the constraints of the dataflow system to allow Salmon to display objects as soon as the first arrive in Salmon. We would also like a module to be able to place objects in the scene that do not need to be recreated each time the module executes.

Due to these usage requirements, it is more natural to cast Salmon as a server in a client-server relationship than in the producer-consumer relationship provided by the dataflow system. Therefore the ports that are used to connect to Salmon have methods such as `addObject` and `removeObject` in a client-server database style. The objects added to the scene persist until they are removed by the client module, or until the client module is disconnected from Salmon. The Salmon module ignores the execute messages

sent from the scheduler, since the Salmon module is continually blocking for input from the client modules.

4.2.2 Fine-grained Dataflow

The `send/receive` pairs shown above are a simple atomic protocol where the entire dataset is transferred at once. However, it often makes more sense to use “fine-grained” dataflow where appropriate [114]. Fine-grained protocols are tuned to the specific layout of the data, such as for receiving slabs of a regular grid for isosurfacing or scanlines for image processing. In such cases the module receives scanlines and sends scanlines for each scanline in the image, but it semantically sends or receives the full dataset exactly once. The port structures in SCIRun are also used for this purpose. The port simply defines alternate protocols for sending and receiving the dataset. An example of this is described in Chapter 8.

4.3 Steering in a Dataflow System

The dataflow mechanism and the modules have been designed to support steering of large-scale scientific simulations. SCIRun uses three different methods to implement steering in this dataflow-oriented system:

1. **Direct lightweight parameter changes:** A variable is connected to a user interface widget, and that variable is changed directly (by another thread) in the module. The iterative matrix solver module allows the user to change the target error even while the module is executing. This parameter change does not pass a new token through the dataflow network but simply changes the internal state of the SolveMatrix module, effectively changing the definition of the operator rather than triggering a new dataflow event. The interface for SolveMatrix is shown in Figure 4.3.

2. **Cancellation:** When parameters are changed, the module can choose to cancel the current operation. For example, if boundary conditions are changed, it may make sense to cancel the computation to focus on the new solution. This makes the most sense when solving elliptic problems, since the solution does not depend on any previous solution.

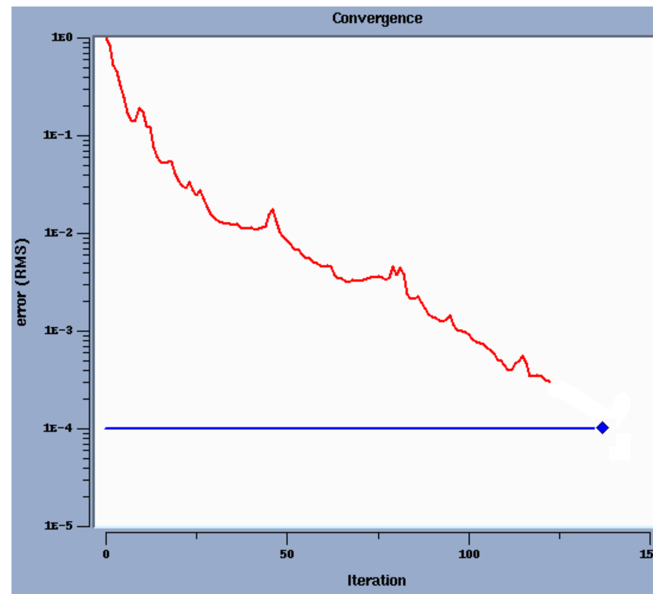

Figure 4.3. A portion user interface for the SolveMatrix module. The user can change the target residual by moving the small diamond on the graph. This is an example of a direct lightweight parameter change.

3. **Feedback loops in the dataflow program:** For a time varying problem, the program usually goes through a time stepping loop with several major operations inside. The boundary conditions are integrated in one or more of these operations. If this loop is implemented in the dataflow system, then the user can make changes in those operators that are integrated on the next trip through the loop. An example of this is shown in Figure 4.4, where the user has created an adaptive finite element solution method. The results of one solution are used to estimate the solution error, and then the mesh is refined in areas of high error. Another solution is computed, and the process is repeated until a target error level has been reached. The user can change the mesh adaptation criteria shown on the left, but the new values are not used until the next iteration of the loop.

4.4 Chapter Summary

We presented the implementation of the dataflow system, including the scheduler. Demand-driven dataflow was presented as an alternate method for implementing a software dataflow system. We have described many of the unique features of the SCIRun sys-

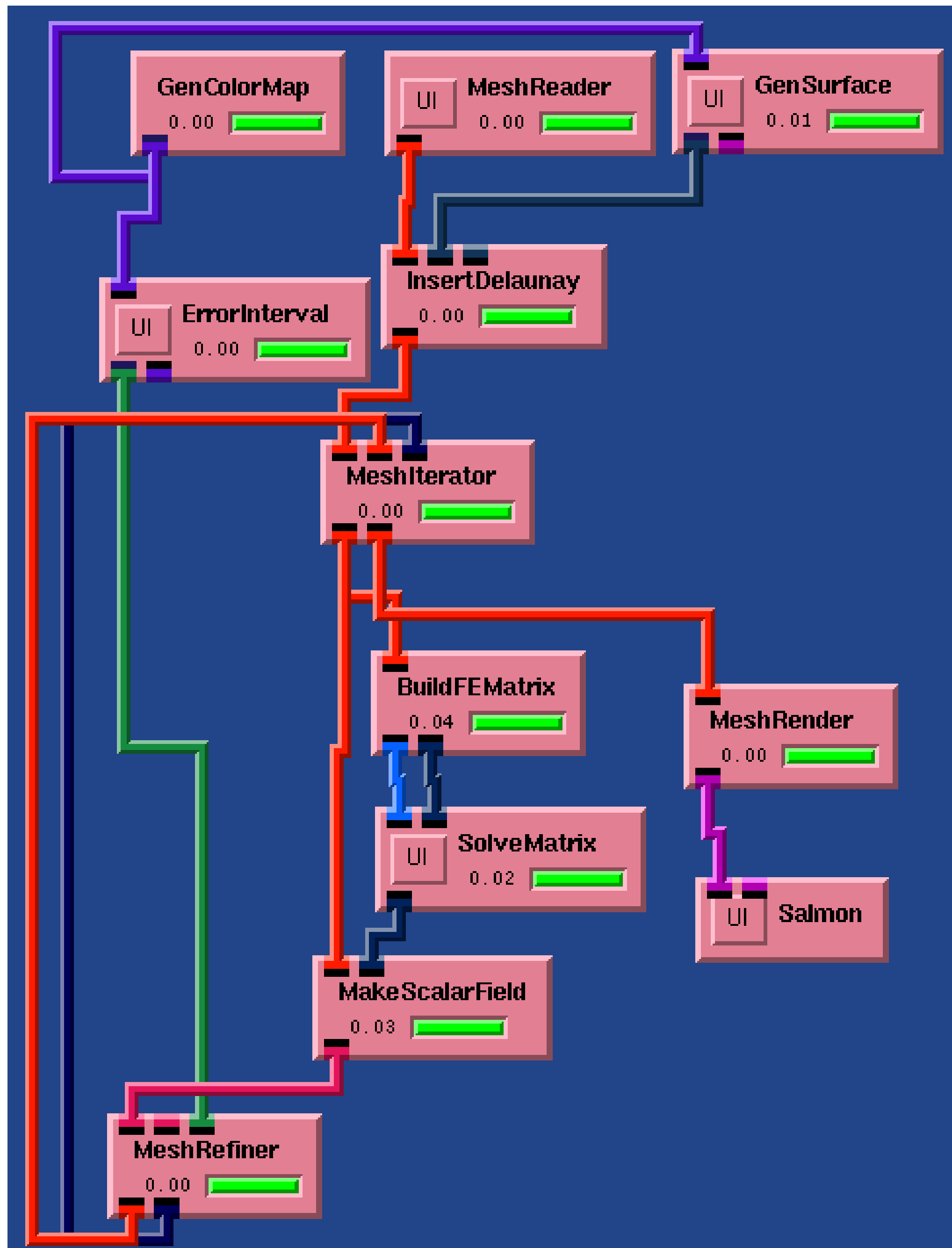

Figure 4.4. Demonstration of steering through a feedback loop in the dataflow network. Data flows beginning with an initial Mesh (generated in InsertDelaunay), performs a computation (BuildFEMatrix and SolveMatrix), refines the mesh according to an error estimate (MeshRefiner), and continues back to MeshIterator. The MeshIterator module will continue iterating the loop until the MeshRefiner module declares that no more adaptation is necessary. The final mesh is rendered using the MeshRender module.

tem. The object-oriented dataflow model in SCIRun facilitates the mixture of demand-driven and data-driven dataflow concepts, which has not been possible in previous software dataflow implementations. A modification of the dataflow ports allow fine-grained dataflow to be intermingled with coarse-grained dataflow in the same simulation. These efficiency features of SCIRun are expressive, allowing a wide range of simulation components to be implemented efficiently. Finally, we describe the methods used to steer simulations in a dataflow environment.

CHAPTER 5

DATA MODELS

In conjunction with the dataflow mechanisms just described, SCIRun derives additional flexibility from object-oriented [35, 68] data representations that are passed from one module to another. These *datatypes*, as they are called within SCIRun, are the basic building blocks for scientific computation. In this chapter, we discuss the different data models and datatypes that SCIRun uses in a wide range of scientific applications. The object-oriented data model is a key component of the computational steering system provided by SCIRun. It provides mechanisms for extending the system in ways not possible with traditional dataflow systems. Specific portions of this data model will be presented to indicate the ways in which they can be used to extend the system.

SCIRun makes use of some features of object-oriented programming to achieve a high level of code flexibility and thus reusability. In an object-oriented data model, pieces of data are thought of as objects upon which computations are executed. A powerful property of objects is that they can be specialized, or *derived,* from a more general object into variants with differing functionality. In SCIRun, a user can easily introduce a new, specialized type of object, without having to alter any other part of the system that uses the same general type. We elucidate this point with the matrix example shown in Figure 5.1. In this example, SCIRun already contains an iterative solver that only requires that the input matrix provides its own matrix-vector product operation. Note that objects contain not only the data values, in this case the contents of the matrix, but also the operations that can be performed on that data. Thus, a matrix object might contain the algorithm required to perform basic matrix operations such as matrix-vector multiplication. If we now create a new derived matrix object, such as one for sparse matrices, then as long as the matrix-vector operation is supported in this new object,

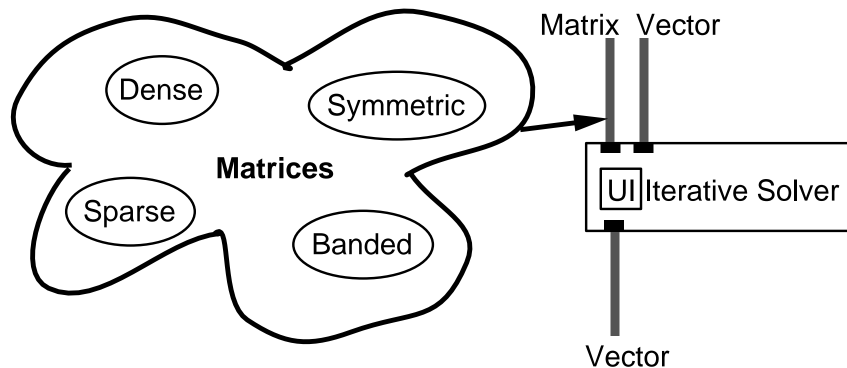

Figure 5.1. Example of flexibility achieved from the SCIRun object-oriented data model. The module shown in the figure uses an iterative method to solve a matrix/vector system ($A\mathbf{x} = \mathbf{b}$). Because all matrix types (dense, sparse, etc.) are derived from the same basic type supported by the iterative solver, only one such solver is required.

any other module written for the basis matrix object type accepts the sparse matrix and continue to function. Thus, within SCIRun a single solver accepts input from any matrix, as long as it is either of the general type, or any type derived from that general type. This level of flexibility ensures maximum reuse of essential elements and allows development efforts to focus on these critical components instead of having multiple versions to create and maintain.

The design of these general types is a very critical component of SCIRun. If the general type requires that a multitude of operations must be supported, then the interface becomes too cumbersome to extend. Similarly, if the operations are complex, then extension is difficult. On the other hand, the operations must be able to efficiently support a gamut of algorithms and data structures. Since these operations require a C++ virtual function call, the program suffers in efficiency if they are called frequently.

5.1 Common Datatypes

A few of the common datatypes used in SCIRun are described here. This is not a fixed set – datatypes may be added by new modules, and the existing set of datatypes may be extended by new modules. In each of the following subsections, we describe the operations that a particular datatype must support, and describe some of the specific types and show how some of these operations would be implemented.

5.1.1 Matrix

The Matrix Datatype is intended to represent a matrix for linear algebra purposes. The Matrix class provides the following interfaces:

- `nrows` - returns the number of rows in the matrix.
- `ncols` - returns the number of columns in the matrix.
- `isSymmetric` - returns true if the matrix is symmetric, false otherwise. Note that some matrix types always return false, even if their contents are actually symmetric.
- `minValue` - returns the smallest value in the matrix.
- `maxValue` - returns the largest value in the matrix.
- `mult` - multiplies a matrix by a vector, and stores the result in a second vector.
- `multTranspose` - multiplies the transpose of the matrix by a vector, and stores the result in a second vector.
- `getRowNonZeros` - returns the nonzero elements for a particular row.
- `zero` - zeros out the entire matrix

The `minValue`, `maxValue` and `getRowNonZeros` operations are primarily intended for matrix visualization, while the rest are oriented towards iterative solution methods. The use of this datatype is described in further detail in Section 7.2.

SCIRun currently implements four different types of matrices. The `DenseMatrix` class stores a single block containing all rows and columns of the matrix. The `TriDiagonalMatrix` class stores three elements per row. The `SparseRowMatrix` and `SymSparseRowMatrix` classes both use the popular compressed row storage format to store sparse matrices. `SymSparseRowMatrix` is the symmetric version of `SparseRowMatrix`. Both store the complete matrix, but `SymSparseRowMatrix` can use the same algorithm for `mult` and `multTranspose`, eliminating the more expensive multiplication that is required for `multTranspose`. Instances of matrix

classes have also be implemented with *SparseLib++* [30], and could also use other matrix packages.

The `mult` and `multTranspose` operations both take an optional beginning and end row argument. This allows the matrix multiplication to be divided among processors in a parallel implementation.

The discussion of the SolveMatrix module displays how these abstract interfaces are used to implement the conjugate gradient algorithm without regard for the actual layout of the matrix. Other operations, such as matrix-matrix multiply, are better implemented with code that checks the actual type of the matrices and uses the most efficient multiplication algorithm and resultant matrix format.

5.1.2 The Mesh Class

The `Mesh` class is not an abstract interface but is meant to be a powerful class for operating on tetrahedral unstructured grids. A Mesh consists of a set of `Nodes` and a set of `Elements`. A `Node` class contains its point in 3D space, and an `Element` class contains a pointer to four different `Nodes`. These data completely specify the mesh, but we also maintain other information for making mesh operations efficient. The full version of the `Node` data structure also contains a list of the `Elements` to which it is connected. The `Element` data structure contains the face neighbors – the `elements` which neighbor its four faces. In addition, the `Element` data structure can optionally contain the element basis function and element volume. The element basis function is a large amount of data, increasing the size of the `Element` data structure from 40 bytes to 176 bytes. This can cause memory limitations for large problems, but it avoids repeated recalculation of these basis functions and so increases the speed of finite element matrix assembly, element interpolation, and many other operations. The user may select at compile time whether or not to maintain the element basis functions.

We could have chosen to make meshes of arbitrary dimension, but we opted for the simplicity and efficiency afforded by hard coding the dimensionality to three. Similarly, we have focused on tetrahedral unstructured meshes. A hexahedral element unstructured mesh has also been implemented. Mixed-element meshes and other dimensionalities will be implemented in the future as applications demand them.

5.1.3 Fields

A field is a scalar- or vector-valued function defined over some region of space. In scientific computing the fields represent some physical quantity, such as voltage, pressure, temperature, flow velocity, etc. The fields data structures provide two base classes: `ScalarField` and `VectorField`. We chose to separate these two types to clarify which operations make sense on which type. These fields are usually approximated with a discrete set of elements. The values are interpolated over the elements in piecewise constant, piecewise linear or possibly higher-order fashion. Tensor fields have been implemented in a similar fashion, but they are not described here.

There are currently several different implementations of the `ScalarField`, including: `ScalarFieldRG`, which defines a scalar field using a regularly sampled grid, and `ScalarFieldUG`, which contains a `Mesh`, and values for each `Node` or `Element`. Figure 5.2 shows the class structure for the `VectorField` classes. The `ScalarField` classes are structured similarly.

The field datatype supports the following operations:

- `getBounds` - returns the bounding box of the defined region.
- `getMinMax` - returns the smallest and largest value in a scalar field (not available for a vector field).
- `interpolate` - returns the value at a specific point. A context allows that query to be efficiently performed on nearby points (such as in streamline advection).

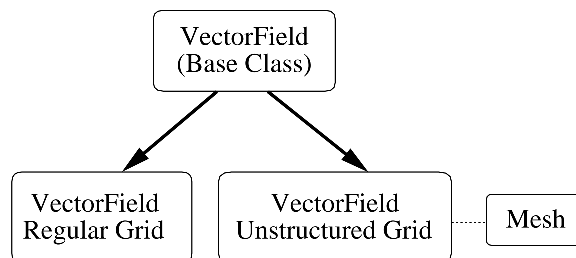

Figure 5.2. VectorField class hierarchy showing the how the unstructured grid and regular grid versions are derived from a common base class. The dotted line to the Mesh shows that the Unstructured Grid version contains a 3D tetrahedral mesh. In addition, the field contains a Vector value for each node or element in the mesh.

There are times when the interpolate query is ineffective or cost-prohibitive. For example, this occurs in the isosurfacing module; the isosurfacing algorithm would be more efficient if the algorithm can take advantage of the underlying data representation [73]. We use general, extensible interfaces wherever they can be used without incurring significant performance or complexity penalties. However, when a general interface is not suitable, we use the run-time typing information (RTTI) facilities in C++ to query the exact type of a particular data object. A module can implement the fast algorithm for the specific types that it knows about and then fall back on a more general algorithm or print an error for types that were not programmed in advance. In practice, this type extraction is seldom needed.

5.1.4 Surfaces Data Structures

There is another class hierarchy that describes a surface in 3D space. For our work, the most important surface is a triangulated surface defined as a collection of 3D points and the corresponding triangles that connect them. These surfaces can be tagged with boundary condition information for later integration with finite element problems. A few other surfaces are provided such as cylinders, spheres, and points (a degenerate surface).

5.1.5 Other Datatypes

There are several more datatypes currently implemented in SCIRun. Some of them are more mature than others, and some of them are more general than others. Some of these are as follows:

1. `Boolean` - provides a simple true/false value for use with feedback loops (iterator modules).
2. `ColorMap` - provides a mapping of data values to colors, used by many of the visualization modules.
3. `ColumnMatrix` - provides a simple column vector for use by the matrix solver.
4. `ContourSet` - represents a set of planar contours, currently used by the LaceContours module.

5. `Geometry` - represents renderable objects. This is described in further detail in Section 7.5.

6. `MultiMesh` - represents a set of multiresolution meshes and is currently experimental.

7. `Sound` - represents a stream of sound, which was used to develop some of the fine-grained dataflow algorithms.

8. `Image` - represents a 2D image.

The datatypes library is a powerful set of data structures for scientific computing, but it also provides a powerful method of extending SCIRun. A typical dataflow-oriented program can be extended by adding new modules to implement new algorithms. However, SCIRun can be extended by extending the abstract interfaces in the datatypes library to operate on other data formats. For example, one can make a new `Field` datatype that implements the `ScalarField` interface for some type of spectral method. Most downstream modules would be able to operate on the data without modifying those modules and without converting and duplicating the data. Note that we are careful to say "most of the time." As mentioned above, there are times when we violate these abstract interfaces for efficiency purposes.

5.2 Persistent Data Storage

SCIRun, like other modern scientific computing codes, uses modern data structures that are generally more complex than a large array of numbers. We needed a facility to save these complex data structures on disk. To solve this problem, we employed the idea of "persistent objects," where complex data structures can be flattened into a data file and then reconstructed into their original form. Since C++ does not support persistent objects automatically, we designed a set of utilities to make these input and output routines in a simple manner.

There are three different levels on which the persistent I/O routines operate:

1. **Primitive:** There is an overloaded global (free) function, `Pio`, for each of the basic

types (float, double, int, etc.), which can both read and write these primitive types.

2. **Simple structure:** For basic structures without class derivations, the user can overload a global (free) `Pio` function to serialize these objects. Usually, this function just calls `Pio` on each of the individual data elements.

3. **Complex structure:** For a complicated class hierarchy, the user writes a virtual `io` function that emits a class name and version number, then the `io` method for the superclass, and then calls other `Pio` functions on the individual components.

The basic stream I/O routines support both binary files and somewhat humanly readable text files. Binary input and output uses the Sun Microsystem's XDR library [47] for portable binary I/O, which allows us to move files between different architectures without the need to convert to ASCII text. Binary files are sometimes smaller and are always a factor of 3-5 faster in reading and writing.

In addition, we support versioning of objects so that we can change the data structures in the code without requiring the conversion of all old data files. The system always knows how to convert from old files, and always writes the current version of the file. The programmer that maintains the `io` method in each class should add code to convert data from older versions, and should provide reasonable default values for data members that have been added in a later version.

To illustrate the simplicity of reading and writing objects in this manner, consider an example of a 3D tetrahedral mesh:

```
class Mesh {
  Array1<Node*> nodes;
  Array1<Element*> elems;
  virtual void io(Piostream& stream);
  ...;
};

#define MESH_VERSION 1

void Mesh::io(Piostream& stream)
{
  stream.begin_class("Mesh",
                     MESH_VERSION); // identify type & version
  Pio(stream, nodes);               // read/write the nodes
```

```
  Pio(stream, elems);                // read/write the elements
  stream.end_class();
}
```

The `Array1` classes know how to read and write themselves:

```
template<class T>
void Pio(Piostream& stream, Array1<T>& array)
{
  stream.begin_class("Array1",
                     ARRAY1_VERSION); // id the type &version
  int size=array.size;                // grab the size of the array...
  Pio(stream, size);                  // ...and write or read it
  if(stream.reading())                // if we're reading...
    array.resize(size);               // ...allocate space for objects
  for(int i=0;i<size;i++)
    Pio(stream, array.objs[i]);       // read/write all of the objects
  stream.end_class();
}
```

and the `Node/Element` classes know how to read and write themselves:

```
void Pio(Piostream& stream, Node*& node)
{
  stream.begin_cheap_delim(); // A delimiter for making text
                              // files easier to read
  if(stream.reading())        // if we're reading...
      node=new Node();        // ...allocate a new node
  Pio(stream, node.pt);       // read/write the node's location
  stream.end_cheap_delim();
}

void Pio(Piostream& stream, Element* elem)
{
  if(stream.reading())        // if we're reading...
      elem=new Element();     // ...allocate a new element
  stream.begin_cheap_delim();
  Pio(stream, elem->n[0]);    // read/write all of the
  Pio(stream, elem->n[1]);    //    indices for the four
  Pio(stream, elem->n[2]);    //    nodes composing the
  Pio(stream, elem->n[3]);    //    tetrahedral element
  stream.end_cheap_delim();
}
```

It is important to remember that these small functions are used to both read and write the mesh for both binary files and text files. This feature virtually eliminates the potential for incompatibilities between the reading code and the writing code. However, the real power comes when we emit a scalar field based on these meshes:

```
class ScalarFieldUG : public ScalarField {
```

```
  MeshHandle mesh;
  Array1<double> data;
  ...;
};

void ScalarFieldUG::io(Piostream& stream)
{
  stream.begin_class("ScalarFieldUG", SCALARFIELDUG_VERSION);
  ScalarField::io(stream); // This serializes the base class
  Pio(stream, mesh);       // read/write the mesh
  Pio(stream, data);       // read/write the scalar data
}
```

Then, we can even emit multiple scalar fields:

```
  Pio(stream, field1);  // read/write field1
  Pio(stream, field2);  // read/write field2
  ...
```

In this example, the fields might share a common `Mesh`. In this case, the `Mesh` object would be written into the file only once. We have omitted many of the details of how this is implemented internally, but the `Pio` routines do not need to be concerned with this mechanism – it is handled internally by the `Piostream`.

We have found that this mechanism is a powerful way to implement I/O for complex data structures. The versioning system allows us to use datafiles that were written years ago without converting them – as long as the programmers maintain the `io` methods when data structures are updated. Using the same code for reading and writing drastically reduces the number of errors in the input/output routines. In addition, this mechanism has increased the utility of binary files by avoiding the need to write a separate I/O function.

5.3 Chapter Summary

We describe the data models used in SCIRun. The use of object-oriented data structures facilitates a new level of extensibility that is not available in previous implementations of dataflow systems. The details of the Matrix, Mesh, and Field abstract classes were described. Finally, we presented a mechanism for storing these complex data structures on disk.

CHAPTER 6

SUPPORT LIBRARIES

This chapter discusses some of the tools that were used to build SCIRun. These tools provide powerful operations with simple interfaces so that they are easy to use when programming. Many of these tools address concerns that are unique to large-scale scientific computing. These libraries are key to achieving high performance with SCIRun, but they do not constitute the original thesis research. This chapter gives a flavor of the issues addressed in implementing a high performance environment.

To implement SCIRun, we have broken down SCIRun into a layered set of libraries. These libraries are organized as shown in Figure 6.1. SCIRun uses an object-oriented design; however, it should be stressed that we have paid careful attention to avoid over-using the object-oriented paradigm to a point that efficiency suffers.

In implementing the SCIRun kernel and modules, we leverage off of a toolbox of C++ classes that have been tuned for scientific computing and operation in a multithreaded environment. We discuss these classes below, starting with the lowest level library and proceeding to more complex libraries.

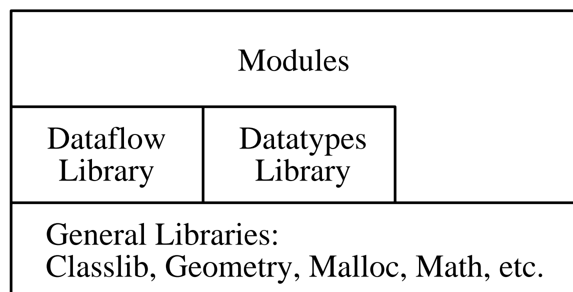

Figure 6.1. SCIRun library organization.

6.1 Malloc, operator new: libMalloc

We have encountered several problems with the implementations of malloc/free and new/delete that are available on current Unix systems. Difficulties with the current Unix implementations of malloc and new include:

1. They are not robust against erroneous behavior. This is particularly confusing when the user's program crashes in malloc, even though the actual error resulted from freeing a bad pointer in a previous call. A multithreaded environment further exacerbates this problem, allowing errors in one thread to cause another thread to crash.

2. They are not thread-safe (reentrant) on many systems. This is typically the case on systems without a native implementation of threads. Accessing malloc and free in such an environment can cause frequent nondeterministic crashes.

3. They do not reveal statistics about their operation.

4. They do not return memory to the operating system when it is no longer being used.

5. They are slow when allocating and deallocating large numbers of small objects.

6. They have a large percentage of memory overhead for small objects.

Of course, the goal would be to resolve all of these problems, but we find that many of the requirements conflict. For example, it is difficult to have bullet-proof behavior against errors without incurring additional overhead, even for small objects.

The implementation of libMalloc centers around the Pool class. Pool defines a constructor and destructor, as well as the methods alloc, free, realloc, get_stats and audit as shown below.

```
class Pool {
   Mutex lock;
   ...;
public:
   Pool();
   ~Pool();
   void* alloc(size_t size, char* ctag, int itag);
   void free(void* p);
```

```
    void* realloc(void* p, size_t size);
    void audit();
    void get_stats(size_t statbuf[18]);
    int nbins();
    void get_bin_stats(int bin, size_t statbuf[6]);
    ...
};
```

Pool represents a pool of memory. At startup, there is a single pool, `default_pool`, from which requests from `malloc` and `new` are granted. The implementations of `malloc` and the `new` operator simply call the `alloc` method of the default pool. Subsequently, the `free` and operator `delete` methods call the `free` method of the default pool. The default `malloc` and operator `new` provide generic information as the two tags for the allocation, but there are alternate interfaces that automatically provide the file and line number for these tags.

The `alloc` method uses three slightly different memory allocation algorithms for small, medium and large objects. Based on heuristics from current applications, small objects are those less than 512 bytes, medium objects range from 513 bytes-64k bytes, and large objects are those over 64k bytes. These ranges are configurable at compile time.

Small and medium objects both use an algorithm based on bins. A bin contains a list of free objects. When free space is requested, `alloc` figures out which bin contains objects of the appropriate size, and the first one from the list is removed. Sentinels are placed at the beginning and at the end of the actual allocation. Small and medium bins differ in how the bins are refilled when they become empty. Small bins use an aggressive fill scheme, where 64k worth of objects are placed in the bin's free list to minimize the number of refills. Medium objects, on the other hand, use a less aggressive scheme – objects are allocated from a larger pool one at a time. Large objects are allocated with independent `mmap` calls to the operating system. This allows the large objects to be returned to the operating system when they are no longer needed. To avoid releasing and rerequesting memory, these large chunks are returned to the operating system (unmapped) in a lazy fashion. It is possible for this policy to fragment the address space of the program, but in practice this has not been a problem and will not be a

problem for 64-bit programs for the foreseeable future.

The algorithms for the three different allocation ranges are based on the philosophy that bigger objects can afford to use more CPU cycles in trying to be efficient, since large objects are allocated less frequently and used for a longer period of time. It is also more valuable to minimize waste for large objects than for small allocations.

To make the pool thread safe, each of the methods acquires the mutex before accessing or modifying any data in the Pool and releases the mutex when these operations are complete. The `alloc` and `release` methods attempt to minimize the time that the pool is locked by performing most operations (tag/header manipulation, verification, etc.) without holding the lock.

This implementation resolves several of the problems described above. Robustness (item 1) has been improved in practice, as the system catches many common errors. Nonetheless, a more specialized memory analysis tool (such as Purify [5]) is better suited to catching severe memory usage patterns. The memory overhead (item 6) is approximately the same as current implementations, and the time overhead for small objects (item 5) is considerably smaller, but still too large. In the Section 6.3, we see a mechanism that may be layered on top of libMalloc to resolve these problems. The other three problems have been resolved.

This memory allocator can also reveal statistics about its operation. Figure 6.2 shows these statistics displayed by a running program.

6.2 The Multitask Library: libMultitask

SCIRun derives much of its flexibility from its internal use of threads [111]. Threads allow multiple concurrent execution paths in a single program. SCIRun uses threads to facilitate parallel execution, to allow user interaction while computation is in progress, and to allow the system to change variables without interrupting a simulation. However, standards for implementing threads are only starting to appear, and the standards that are appearing are, thus far, cumbersome.

Fortunately, the facilities provided by various thread libraries are similar. Synchronization primitives which do not exist in one library can be constructed from the synchronization primitives that are available. libMultitask is a layer that provides a simple,

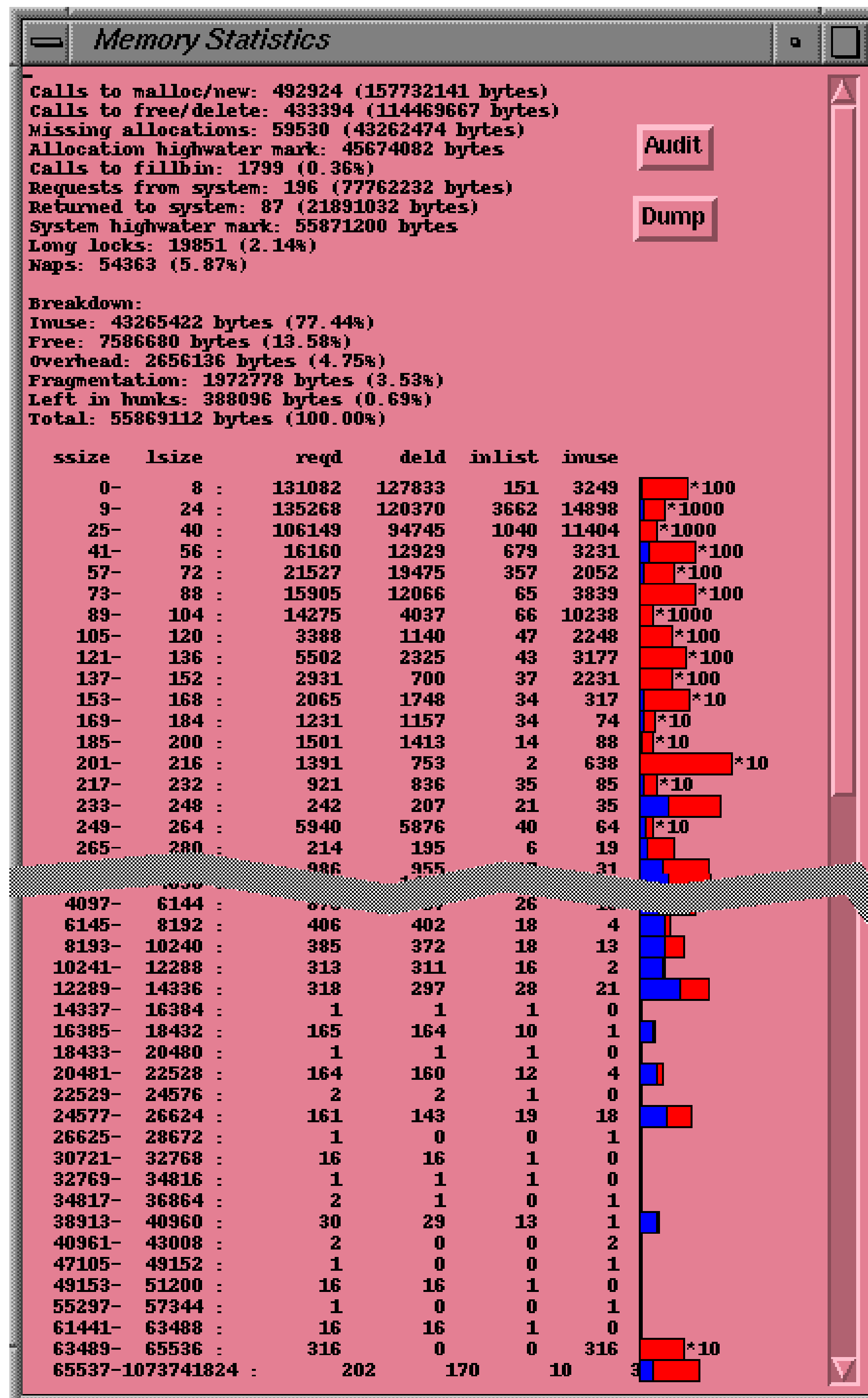

Figure 6.2. A portion of the statistics of the custom allocator, showing bytes allocated and freed, high water marks, and spinlock statistics. Some of these statistics are also displayed for each bin. To the right of each bin a small graph shows the objects in the freelist and the objects in use for each range of sizes.

clean C++ interface to threads and provides abstraction from the actual standard used to implement them.

6.2.1 Tasks

The Multitask library provides a class `Task`, which encapsulates a thread. The user creates a subclass, derived from Task. The Task constructor requires a name for the Task and a priority. A new thread is created when the creator calls the `activate` method of the Task class, which causes the (overloaded) `body` method to be started in a separate thread. `Activate` returns immediately and will not wait for `body` to complete - thus triggering concurrent execution in the program, similar to a `fork()` operation. However, all of the threads (Tasks) share access to a common heap – unlike the traditional `fork()` function. Task is an abstract base class because it does not actually provide a `body` function. Other classes inherit from Task, providing a `body` function to do the actual work of the thread. The thread continues until the `body` function returns, or until `Task::exit` is called.

Task also provides static functions to return the number of processors available on the system, to start-up multiple threads for a function, and to cause all threads to exit.

6.2.2 Intertask Communication

The Multitask library also provides a number of convenient synchronization primitives for these tasks to communicate with each other – Inter-Task Communication (ITC). ITC primitives are as follows:

- `Mutex` provides `lock`, `try_lock` and `unlock` methods.
- `Semaphore` is a counting semaphore, providing `down, try_down` and `up` methods.
- `Barrier` provides a single `wait(int nwait)` method to allow a group of threads to stop executing at the barrier until all `nwait` threads arrive.
- `ConditionVariable` provides `wait(Mutex& lock), cond_signal` and `cond_broadcast` methods.

- CrowdMonitor, a multiple-reader, single-writer access control primitive, provides read_lock, read_trylock, read_unlock, write_lock, write_trylock and write_unlock methods.

- Mailbox, a fixed-length, thread-safe FIFO (First-In, First-Out communication pipe), allows multiple senders and multiple receivers. This is a template class that provides send, try_send, receive and try_receive methods. The mailbox allows multiple threads to send tokens to the mailbox and an arbitrary number of threads to receive tokens from the mailbox. These tokens are typically pointers to a message structure. Using this primitive, one can implement threads that behave like a small server, with other threads acting as clients. The dataflow mechanism described in Chapter 4 uses this mechanism to communicate between the scheduler and modules, and the modules use it to send data through the ports.

- Other structures are AsyncReply, which provides a single pigeon hole rendezvous point and classes to perform reduction operations.

The Task and ITC methods have been implemented in four different environments: SGI IRIX (using sproc and us primitives) [113], with Posix threads [20] (aka pthreads), with Solaris threads [71], and with setjmp/longjmp [7]. They all provide similar mechanisms. The setjmp/longjmp implementation is not preemptive but could be made so if necessary. The sproc version provides heavyweight threads, since each thread is a different share-group process. This allows the threads to be debugged easily using a traditional debugger, but one must be careful to limit the number of threads used. Since the pthread standard has now been approved, it will probably be the preferred implementation for the future. However, libMultitask or a derivative of it (see Appendix) will likely remain as a convenient interface to Posix threads.

6.3 Generic Tools: libClasslib

This is a collection of various tools that are valuable in constructing SCIRun's kernel and computational modules. Some of these data structures overlap those available in the Standard Template Library (STL) [84], but our implementation predates common accep-

tance of STL. In implementing these, we have not tried to make extravagant general-use interfaces. Rather, we have designed our tools to be simple, easy to use, and efficient. We have also designed these interfaces to perform the operations that we require, avoiding the temptation to over-engineer them. Data structures that we have implemented include an unbounded queue, a bounded stack, a dynamic hashtable, and a dynamic array class. These structures use templates to make them usable as containers for any type.

6.3.1 TrivialAllocator

Another useful tool is the `TrivialAllocator` class. This class is designed to increase the efficiency of the `new/delete` operator for small objects that are allocated and deleted frequently. The TrivialAllocator simply keeps a list of free objects that are ready to be used. Using the TrivialAllocator for a particular class simply requires redefining the operator `new` and operator `delete` methods to call the `alloc` and `free` methods of the TrivialAllocator. Using the TrivialAllocator class is significantly more efficient than using the general operator `new`, because most of the time it simply returns the first item off of the free list. Objects are allocated in groups, using the second parameter to the constructor as the number of objects to be allocated at a time. The TrivalAllocator `free` method always just puts the object back on the free list. The free list is accessed in a last-in/first-out manner to maximize cache locality. Since these are used in a multithreaded environment, `alloc` and `free` both require the acquisition and release of a mutex. However, this is a separate mutex from the global allocator, so it is not be subject to the same contention.

This tool allows us to work around the per-object overhead and allocation time required for small, high-use objects. However, it does so at the expense of the overrun detection and consistency checks that our implementation of `new` and `delete` provide. A future implementation will provide a mechanism by which trivial allocators can be disabled through a debug environment variable – reducing run-time performance, but allowing the consistency checks to be made.

6.3.2 Handles and LockingHandles

`Handles` are a "smart pointer" mechanism for automatically maintaining reference counts in objects [116]. SCIRun uses these to facilitate the sharing of large data structures between modules. The last Handle to "let go" is responsible for deleting the object. Reference counting provides a simple form of garbage collection, without the high overhead and unpredictability associated with a full garbage collection system. The largest weakness is that reference counting can fail to destroy objects that contain circular references (including circular lists, self references, and back pointers). However, it does provide the advantage that objects are destroyed immediately when the last handle releases the object. This feature is essential for large scale scientific computing where the memory resources held by such a handle need to be carefully controlled.

A Handle contains a single data member `rep`, which contains a pointer to the actual representation. It also defines constructors, a destructor and accessor methods that increment and decrement a reference count in the object. The objects used in a handle must provide a member called `ref_cnt`, which is initialized to zero at construction time. In addition, objects that support the `detach` operation must support a `clone` method which duplicates the object. Since template syntax is sometimes rather clumsy, it is often convenient to typedef a "smart pointer" type for different object types, such as:

```
typedef LockingHandle<Object> ObjectHandle;
```

A `LockingHandle` is similar to Handle, but with each object also providing a Mutex member that is locked for all of the `ref_cnt` operations.

6.3.3 Timers

Two convenient classes are the `CPUTimer` and the `WallClockTimer`. These provide a stopwatch interface to acquire time information. Methods include `start`, `stop`, `clear`, and `time`. The `time` method simply returns the total accumulated time between `start` and `stop` pairs. Typically, these timers access the Unix timer functions appropriate for the operating system, but they can also utilize a high resolution user-space mappable timer on systems that support it.

6.4 Geometry Library

libGeometry provides `Point`, `Vector`, `Plane`, `Ray`, `Transform`, and `BoundingBox` classes for convenient computation of 3D geometry. The addition, subtraction, and multiplication operators have all been implemented to allow these components to be used in a convenient fashion. Since SCIRun has been designed primarily for 3D computational field problems, the geometry library implements only 3D points, instead of allowing arbitrary dimensional geometry. This was a sacrifice of generality for simplicity and efficiency.

We have chosen to separate the concept of a `Point` from the concept of a `Vector` [28, 29]. For the sake of efficiency, these are both specialized for three dimensions, with an `x`, `y`, and `z` component. A `Point` differs from a `Vector` only in the operations that can be performed on it. A `Point` indicates location in 3D space, and a `Vector` indicates offset. A `Point` subtracted from another `Point` produces a `Vector`, a `Vector` added to another `Vector` produces a `Vector`, and so forth. A cross product is defined only for `Vectors`, since they do not make geometric sense for `Points`. This has proven to be a useful way to help the programmer reason about geometric transformations and to write correct code for geometric manipulations.

6.5 The Math Library

libMath provides a somewhat eclectic collection of various core numerical functions, such as `Min`, `Max`, and `Abs`, which have all been overloaded for various numerical types. libMath also contains several core linear algebra loops, such as dot products, vector-vector multiply, etc. Several of these kernels have been highly tuned to take maximum advantage of various architectures.

Some of the functions include the following:

- Thread-safe random number generators.
- A templated spline class to create splines of arbitrary types.
- A complex number class.
- A tuned Fast Fourier Transform.

- A tuned matrix-vector multiply for sparse matrices.
- A highly tuned vector-vector dot product.
- Utility functions for computing min and max of two or three variables.
- A tuned tri-diagonal matrix solver.
- Other tuned matrix and vector operations.

Other numerical libraries have also been used with SCIRun, including LAPACK [12], SparseLib++ [30], and portions of Diffpack [69].

6.6 Chapter Summary

We presented an overview of the low-level libraries used in SCIRun. A replacement for malloc is used to overcome difficulties with the previous implementations in a complex multithreaded environment. A multitasking library is used to simplify the use of threads and synchronization constructs, and also provides an insulating layer above different thread implementations. Two general toolbox classes provide generic data structures and computational algorithms used throughout the SCIRun system and SCIRun modules. Handles were described as a “fool-resistant” mechanism for managing the destruction of data in SCIRun.

CHAPTER 7

MODULES

In Chapters 4 - 6 we have described a set of computational tools and a dataflow system for imposing control structure. However, the real value in SCIRun comes from how these components are leveraged in applications implemented via SCIRun modules. This section covers how a module is constructed and then describes a few of the key modules in SCIRun, showing how they use the features that we have described above.

7.1 Writing a Module

The process of writing a new module involves writing a new C++ class. The constructor for this class creates the input and output ports for the module and defines parameters that the user interface may control. A single virtual function, `execute`, is overloaded to perform the actual work of the module. The `execute` function typically receives data on the input ports, performs some computation and then sends data on the output ports. Other callback functions can provide input from user interface components and 3D widgets. Adding a user interface to a module involves writing a small Tcl [88] script that sets up the components of the interface.

Some modules may not have input ports, and others may not have output ports. A module with only output ports is called a source, a module with only input ports is called a sink, and a module with both is called a filter.

Existing code may be integrated into SCIRun by writing a small wrapper module that passes data into and out of an existing C, C++ or Fortran program. The wrapper module may be required to translate data structures before passing them to the existing code and before sending them down the pipe. To avoid this translation, existing code can also be incorporated by extending the class hierarchy that flows through the dataflow network. For example, instead of translating to a specific ScalarField class the module can send a

new subclass of the ScalarField that would provide `interpolate` and other methods for use by downstream modules.

SCIRun was originally designed as an environment for developing new simulations and computational components. We are currently working on ways to more automatically incorporating existing packages into the SCIRun visual programming environment. Currently, one can integrate existing code into SCIRun in two different ways. One way is to implement a module that uses existing code to perform operations on SCIRun datatypes. The second is to implement an Adapter [35] object that allows other SCIRun modules to operate on data generated by the third party modules.

7.2 FEM and Matrix Modules

The BuildFEMatrix module takes a tetrahedral mesh and builds a stiffness matrix for a finite element approximation. The current version of this module approximates the generalized 3D Poisson equation (or its homogeneous counterpart, Laplace's equation) in the discretized physical domain Ω:

$$\nabla \cdot \sigma \nabla u = f \text{ in } \Omega \,, \tag{7.1}$$

where f is a vector of sources and σ is 3×3 tensor corresponding to the materials within the domain. The scalar field u is subject to the boundary conditions:

$$u = u_0 \text{ on } \Gamma_1 \text{ and } \sigma \nabla u \cdot \mathbf{n} = 0 \text{ on } \Gamma_2 \,, \tag{7.2}$$

where Γ_1 and Γ_2 represent boundaries within the volume domain, Ω.

The mesh elements have been tagged with a corresponding tensor property (σ) and boundaries have been tagged with the appropriate boundary condition. The module makes two passes – first to build the sparse structure of the matrix using a compressed row storage scheme, and second to fill in the resulting sparse matrix. Building the matrix is performed in parallel to take full advantage of multiple processors. The module also builds the right hand side (load) vector. Users who wish to use other governing equations can extend this module to build the stiffness and load matrices appropriately.

Usually, these matrices are passed into the SolveMatrix module, which uses direct or iterative schemes to find the solution, x to the matrix equation:

$$Ax = b. \tag{7.3}$$

The SolveMatrix module graphs the convergence of the residual as the algorithm iterates. The algorithm stops when the residual is less than some target level (i.e., the solution has converged). The user may move the target residual up or down while the solution is still in progress, thus steering the computation in a simple manner. Figure 7.1 shows the interface for this module in operation.

SolveMatrix uses the `mult` and `mult_transpose` virtual methods in the Matrix base class to perform each iteration. As a result, the user can implement a new matrix type that SolveMatrix can be used without even recompiling the module. This new matrix type does not even need to explicitly store the matrix – it can build the matrix dynamically or implicitly.

SolveMatrix also has an option to use the previous solution as the initial guess for the iterative solver. When small changes are made in the boundary conditions (a common case in an engineering design scenario), the system converges rapidly. This is one instance where an integrated system can actually be more efficient than implementing separate programs for each of the phases.

After each iteration, the module updates various parameters, such as the maximum allowed residual, preconditioning method, and even the choice of solution method. The maximum allowed residual can be updated without affecting the progress of the solver. However, if the preconditioning or solution methods are changed during the course of a solution, the solution is restarted using the partially converged solution as a new starting point. Switching solution methods during the course of a matrix solve will not introduce errors into the final solution. The least significant digits of the solution may differ, but only within the tolerance allowed by the maximum residual. Changing the solution methods during the course of a matrix solution allows the user to experiment with different methods, or to intervene if a particular is converging slowly, or is diverging.

The VisualizeMatrix module draws a figure representing the nonzeros in a matrix. Non-zero entries are drawn with small red dots, and zero entries are left black. This gives the user a quick representation of the sparse structure of the matrix, as shown in

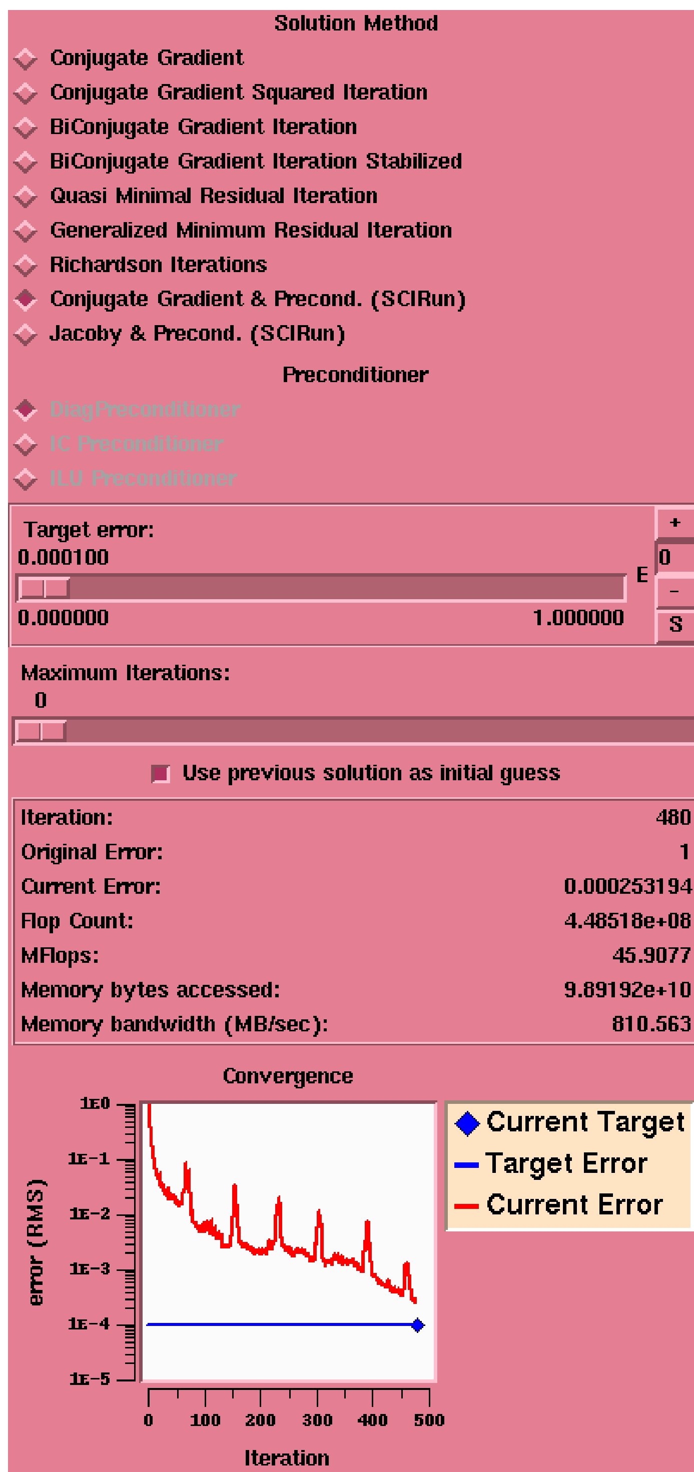

Figure 7.1. The user interface for the SolveMatrix module, showing the iterative methods available, the graph of the convergence and the target residual, which may be changed while the computation is in progress.

Figure 7.2. A magnifying glass can reveal the actual numbers in a portion of the matrix by clicking the mouse in the desired region. VisualizeMatrix uses two additional virtual functions in the Matrix base class: `getRowNonZeros`, which returns the nonzeros in a particular row, and `get`, which returns the number in a specific row and column of the matrix.

7.3 Readers and Writers

The Reader and Writer modules are straightforward. They simply call the Persistent object `io` routines that were described above in Section 5.2. The readers read in a text

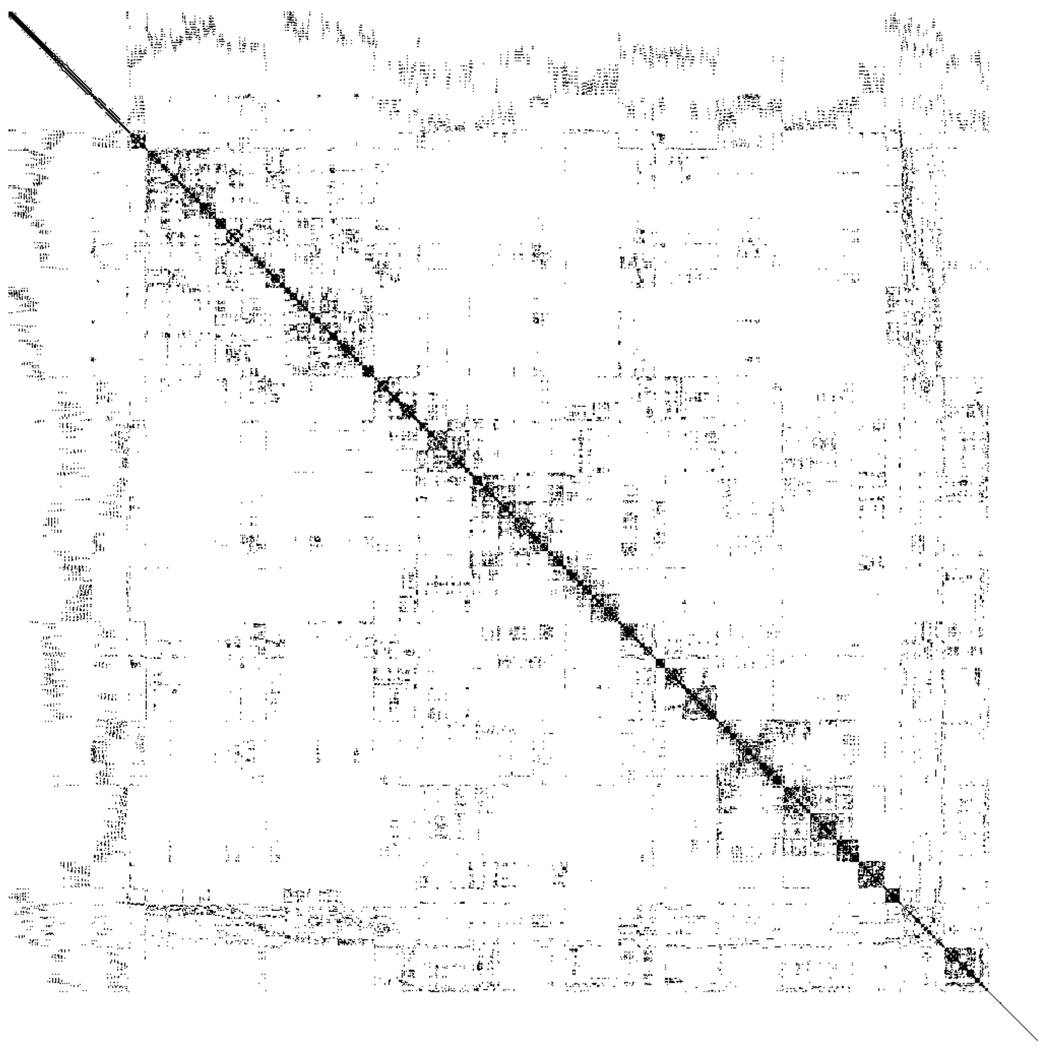

Figure 7.2. Visualization of the sparse structure of a matrix. The black dots represent the nonzeros in a symmetric matrix with approximately 10,000 rows and columns.

or binary persistent object file and send the resulting object downstream, and the writers receive an object and then write it out to disk. Since the support for these modules has been provided by the lower layers, these modules do nothing more than provide user interfaces for the filenames. These modules are automatically generated, but provide hooks for adding user-defined reader/writer functions.

7.4 Visualization Modules

While implementing SCIRun's datatypes, we considered what type of operations SCIRun modules would require. One example of such a consideration is the implementation of the vector and scalar field datatypes. Each of these fields comes in a variety of flavors, corresponding to the internal representation of the data – implicit, explicit, parametric – and the topology of the field – structured (implicit) or unstructured (explicit). The field datatype uses several generic operators to allow module writers to access information from the field without needing to know how the field is internally represented. As discussed above (see section 5.1.3), these operators can query the field for minimum and maximum scalar values or geometric bounds, or for the field's value at an arbitrary point in space. This last operation, retrieving the value of the field at any location, is implemented by an interface called `interpolate`. In the following subsection, we discuss how we have exploited this generic operator in several of our modules, and we provide an example of when we chose to side-step this abstraction to write a more efficient algorithm based on the specific internal representation of the field.

Many of SCIRun's modules exploit the field interface `interpolate` described in Section 5.1.3. The `interpolate` method takes a point as an argument and calculates the value of the field at that specific location. It accomplishes this by determining which element of the field contains the point and linearly interpolating the values at the element's vertices.

One example of a module that calls the `interpolate` method is the Streamline module. The Streamline module is used for vector field visualization: by tracing particles advecting through the vector field, the user can examine local flow phenomena around critical points (such as vortices and turbulence) while also gaining a global sense of

the field's flow. Figure 7.3 shows an example of the electrical current flow near the heart as computed by the Streamline module. We compute the paths of particles through the vector field as discrete line integrals, each corresponding to a streamline. We have several implementations of this integration; one of these is a fourth-order Runge-Kutta method [18]. Given a point p, we integrate along the path to find the next point along the streamline, pNew. The following piece of code accomplishes this:

```
Vector f1, f2, f3, f4;}
Point p1, p2, p3, pNew;

vfield->interpolate(p, f1);
p1 = p + (f1 * 0.5);
vfield->interpolate(p1, f2);
p2 = p + (f2 * 0.5);
vfield->interpolate(p2, f3);
```

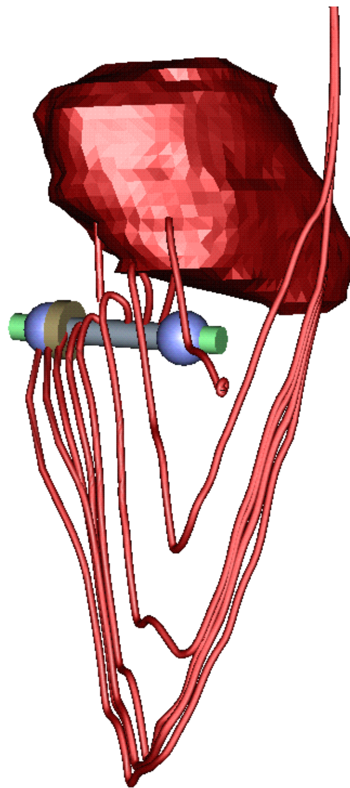

Figure 7.3. Visualization of a defibrillator design simulation, showing an electrode, the surface of the heart (epicardium), a 3D widget (rake), and electrical current lines (streamlines). The other electrode in the simulation is obscured by the heart. Much of the current leaving the visible electrode travels away from the heart, due to the high conductivity of blood in a nearby vessel.

```
p3 = p + f3;
vfield->interpolate(p3, f4);
pNew = p + (f1 + f2 * 2.0 + f3 * 2.0 + f4) / 6.0;
```

The principal idea of this algorithm is that we find the vectors corresponding to particular points near `p` and take a weighted average of these vectors to determine our next location along the streamline. Note that this algorithm works for any type of vector field that implements the `interpolate` interface. By designing a generic interface to the `interpolate` method, we have abstracted away the details of how interpolation is implemented for different field types, and consequently the module writer needs not be concerned with what *type* of vector field is generating the streamlines.

Another module that uses the `interpolate` interface is SurfToGeom. This module takes a surface, a scalar field, and a colormap as input, and outputs colored geometric primitives that can be passed to a renderer. The module computes these primitives in the following way: it queries the field for scalar values corresponding to points on the input surface; these points are assigned material properties by indexing the resultant scalar values in the color table; and finally, these colored points are grouped into geometric primitives (such as triangles or tri-strips). Here again the SurfToGeom module utilizes the `interpolate` interface to find the scalar values of the points on the surface.

The person programming the SurfToGeom module does not ever have to worry about the type of the incoming scalar field - structured, unstructured, or any other type. As long as the `interpolate` interface is provided, the SurfToGeom module works. Furthermore, as new scalar field types are implemented in the future, they automatically work with the SurfToGeom module as long as the `interpolate` method is implemented, without requiring code to be rewritten or even recompiled.

In contrast to the Streamline and SurfToGeom modules, the IsoSurface extraction module does not use the `interpolate` method to find values at points within the field. Although the isosurfacing algorithm could have been implemented this way, we were able to dramatically improve the speed of our algorithm by taking into account (and exploiting) the underlying structure of the data. For example, we have implemented the Marching Cubes algorithm [73] as one option for isosurface extraction. For the

case of both structured and unstructured grids, it is much faster to solve the "forward problem" - finding the intersection of the field with each element than it is to solve the "inverse problem" - using various calls to `interpolate` to locate the isovalue points within the field. In this case, it would have been highly inefficient to use the generic `interpolate` operator, so we wrote data-structure dependent code to solve the problem. The IsoSurface module lacks the abstraction (and as a result, the cleanliness) of the Streamline and SurfToGeom modules. As other field types are implemented, we have to write new isosurfacing code to handle each field type. In this case, there is considerably more to be gained by using data-structure specific information in our implementation than there is to be gained by using the generic accesses permitted by abstracting this information away.

7.5 Salmon Module

One of SCIRun's key modules is the graphical viewer called Salmon. Salmon is named for its ability to spawn multiple views (Roe), and its ability to send messages upstream in the dataflow network. Salmon collects the geometric primitives from any number of modules and presents them in a single 3D view. The user can rotate, scale, and translate the objects, as well as manipulate lighting, camera parameters and rendering method, to obtain the desired view. Other views can be spawned to separate windows to simultaneously display the objects from other viewpoints or with different subsets of the objects.

Geometric primitives are passed from the modules to Salmon as a subset of a scenegraph. These scenegraphs are a tree-based display list that define geometric primitives, colors, and other rendering parameters. Drawing the scene involves traversing the graph and emitting OpenGL commands at each node. Scenegraphs can be composed, stored to disk via the persistent object mechanism, and then read back for later display. Parameters in the scenegraph can be changed dynamically by the module using the CrowdMonitor (multiple reader, single writer) lock described above in Section 6.2.

In addition to using the Salmon module for visual output, we can also use it for 3D input by allowing the user to interact with specific objects in the scene. These objects,

called Widgets [96], allow the user to intuitively augment parameters directly in the 3D scene. For example, to provide the starting point for a streamline advection, the user simply drags a SphereWidget around in the scene. This interaction is generally more intuitive to a user then typing in numbers or manipulating 2D sliders. The user can also use other 3D widgets to specify streamline starting points, such as a rake [26, 44].

7.6 Other Modules

There are many other modules in SCIRun, including the following:

- **FEMError** - computes the upper and lower error bounds of a finite element simulation [132, 133]. This module can be used in a feedback loop, similar to Figure 4.4 to implement an adaptive finite element simulation.

- **MeshRefiner** - uses the Error fields described above to decide where to add new nodes and remove old nodes in order to refine and derefine the mesh.

- **Gradient** - computes a vector field that is the gradient of the given scalar field. The Gradient module is often used in conjunction with various vector field visualization tools, such as Streamline and Hedgehog in order to reveal the underlying processes that produce the scalar field.

- **Magnitude** - computes a scalar field that is the magnitude of the given vector field. This allows scalar field visualization tools to be used to view a vector field, perhaps in conjunction with vector field visualization tools as well.

- **FFTImage / IFFTImage** - takes the Fast Fourier Transform (FFT) and inverse FFT of an image, producing another image which is the frequency space representation of the data.

- **FilterImage** - multiplies two images together to perform a filter operation in frequency space.

- **MeshView,** an interactive tool for manipulating and visualizing 3D tetrahedral unstructured meshes.

- Modules corresponding to various pieces of the Diffpack [69] system.
- Wrapper modules for various software libraries, such as LAPACK [12] and Volpack [8, 67].
- Wrapper modules for various applications, such as Tetrad, described in Chapter 8.

Even this is not an exhaustive list. Chapter 10 mentions components and applications that are currently in progress. Despite the number of modules available, a user often needs something different than what is supplied. In such cases, the user can simply create a new module (by writing a small C++ program). User modules are dynamically linked with SCIRun, which avoids the need to relink the SCIRun executable.

7.7 Performance

Throughout this dissertation, we have advocated the use of object-oriented programming techniques for implementing scientific applications. However, one must be careful to make judicious use of object-oriented abstractions to avoid reducing the performance of the application. In a similar vein, one must be mindful of the performance impacts of extracting and visualizing data from a running computation.

In a representative application, the torso defibrillator model described later in Chapter 8, the time spent by the visualization tools was small compared to the time spent performing the computation. In a simulation consisting of 200,000 nodes and 1.2 million elements, generating the mesh consumed 15 seconds (1.3% of the total), building the finite element matrix consumed 204 seconds (18.2%), solving the matrix consumed 817 seconds (72.8%), and all of the visualization tools combined consumed only 86 seconds (7.7%). These times were reported on a 14 processor SGI Power Onyx, with 195 Mhz R10000 processors. These time are CPU seconds, and due to parallelism the wall clock time was significantly less – about $2\frac{1}{2}$ minutes. Disabling the visualization tools reduced this time by only a few seconds. Using fewer visualization tools can also reduce the time, and likewise using a large number of them can potentially slow the simulation more severely. In this case, there were five visualization tools: a cutting plane, a streamline tool emitting 20 streamlines, two color mapped surfaces, and an isosurface. The IsoSurface

module was the most expensive visualization tool, consuming 50 seconds (4.5% of the total) of CPU time. This represents 58% of the total CPU time spent in the visualization tools. Since the isosurface is being recomputed several times during the course of the computation, it is not practical to employ faster isosurfacing algorithms [72] that require a costly preprocessing stage.

Measuring the abstraction penalty imposed by the object-oriented paradigm is a more difficult problem. Short of reimplementing the entire simulation without these abstractions, one can only measure the CPU time required by those functions. C++ features such as function inlining and templates can help to reduce these overheads. In the same application, these functions represent less than 1% of the CPU time spent on the application. There are cases where the abstraction penalty would be significantly more severe. In Chapter 5, we described a scenario where SCIRun uses the run-time typing information (RTTI) facilities in C++ to avoid the severe penalties of the abstraction. Future research may discover new abstractions or new methods of implementing these abstractions which can be more efficiently used in these scenarios.

7.8 Chapter Summary

We described the process used to write new SCIRun modules, and presented the design of several such modules. Modules used for the finite element method demonstrated how the object-oriented Matrix classes described in Chapter 5 are used. An overview of the visualization tools describe how the Field classes described in Chapter 5 are leveraged to allow the visualization of arbitrary . We also describe how the C++ type system is used to provide efficient implementations of algorithms that cannot be efficiently implemented using the abstract interfaces of the SCIRun object-oriented data model. The key module for many SCIRun applications is the Salmon graphical viewer module. It provides an interactive 3D view of a collection of visualization tools and steering inputs. It facilitates interaction with the simulation through 3D widgets. Finally, we discuss the performance of the object-oriented techniques described in the last three chapters. When managed properly, the performance impacts can be minimal.

CHAPTER 8

RESULTS AND DISCUSSION

The bulk of the "results" from this dissertation are evidenced by the applications that are made possible in SCIRun. The next sections outline four such applications, but many more are possible. The applications presented here represent a broad range of the potential uses of SCIRun. Large scale, iterative, multistep scientific design applications are most suited to the large-grain data-flow multithreaded SCIRun environment.

The first application, a model of the electrical activity in the human torso is modelled with an elliptic partial differential equation, using unstructured grids for complex geometry and material properties. The second involves approximating the rendering equation using Monte Carlo methods, exhibiting a very long-running program. The third application demonstrates CFDLIB, a legacy Fortran program that can simulate a variety of computational fluid dynamics problems. Finally, an atmospheric diffusion example illustrates the incorporation of a time-varying adaptive unstructured grid program in the SCIRun environment.

8.1 Application: Torso defibrillator modeling

As described in Chapter 1, The traditional method for solving bioelectric field problems uses multiple, nonintegrated computer programs. For example, a scientist using a computer simulation to examine the effect of electrode patch placement on transcardiac current density [106] would require geometric modeling, numerical simulation, and scientific visualization tools to complete the task. One program might be used to define the thoracic surfaces from medical images, another to create a discrete mesh of the volume contained within the surfaces [104]. Another application might be used to run a finite element simulation of the electric current distribution from the defibrillation electrodes through the thoracic volume [106]. Another approach might be to write a

Fortran program to create the finite element matrices, and then solve the system of equations using a public domain numerical library such as LAPACK [12].

To see the output would require a scientific visualization package (such as AVS, vtk, and others described in [11]). Between each of these steps, it would be necessary to save the output of one program in a format that the next in the sequence can read—this might necessitate separate file format conversion utilities. To find the optimal location, shape, and size parameters for the defibrillator electrode, the scientist would have to go back to the geometric modeling package, change the necessary parameters, manually rerun all of the subsequent steps to see how the new electrode configuration affects the current density distribution, and then manually iterate. The manual intervention required to drive this process is both tedious and time-consuming.

Far more efficient is a scenario in which the user can define an appropriate set of parameters for a given simulation, and then set up a sequence of runs to examine each of them and save the results for subsequent examinations. The complete execution of the sequence might require hours or even days, but the user would be free during that time to perform other tasks. This process is similar to the "what if?" analysis that modern spreadsheet programs offer for much simpler problems.

In our example of the defibrillation simulation, the scientist can select various locations and orientations for the defibrillation electrodes, choose values for the other parameters of the simulation (e.g., the number of nodes in the finite element model, the boundary conditions, the error tolerance for convergence, and the evaluation criteria), and leave the simulations to run as long as necessary. Viewing the results might be as simple as watching the animation produced by the simulation or scanning other defibrillation quality indices such as maximum and minimum current density magnitude or current density histograms from the heart. This automated execution process, whereby the user selects all of the parameters in advance and does not control the intra- or interpackage execution, is termed *batch processing*. A primary benefit of batch processing it that it allows the scientist to utilize computational resources without the need to continuously guide the process. However, with some computer programs execution cannot be automated. That is, the package cannot be run without regular user intervention

during execution. This constraint makes it impossible to run multiple computational jobs automatically leaving the user with the task of manually initiating and controlling each step of the process.

Here we address the application of our computational steering model to two problems in electrocardiography, the forward (or direct) ECG problem and simulation of cardiac defibrillation. Mathematically, these problems are governed by the generalized Laplace's equation for electrical conduction in the physical domain Ω [94]:

$$\nabla \cdot \sigma \nabla \Phi = 0, \tag{8.1}$$

where σ is an electrical conductivity tensor. The electrostatic potential Φ is subject to the boundary conditions:

$$\Phi = \Phi_0 \text{ on } \Gamma_1 \text{ and}, \sigma \nabla \Phi \cdot \mathbf{n} = 0 \text{ on } \Gamma_2, \tag{8.2}$$

where Γ_1 is the surface of the heart or internal defibrillator electrodes, and Γ_2 is the surface of the torso. Φ_0 specifies a Dirichlet boundary condition on this surface. With $\mathbf{n}$ as the unit surface normal vector, $\sigma \nabla \Phi \cdot \mathbf{n}$ represents the current flow normal to the surface of the torso, which is zero for the insulated boundary.

Once the electrostatic potentials are known, one can calculate the current density $\mathbf{J}$ according to:

$$\mathbf{J} = -\sigma \nabla \Phi. \tag{8.3}$$

The forward ECG problem is characterized by solving equations (8.1-8.3) with Φ_0 equal to voltages measured on the heart's surface and yields information regarding the voltages and current flow within the thorax as a function of the endogenous fields of the heart. For the defibrillation problem, electrodes are either implanted internally or applied directly to the chest to deliver sufficient electric energy to stop the irregular heart rhythms that signify a fibrillating heart [70, 83]. Mathematically, this can be posed as solving equations (1-3) with the voltage boundary condition applied on a portion of the torso boundary, $\Sigma \subseteq \Gamma_2$ for external defibrillation or from the surface of the defibrillation electrode(s) within the volume of the thorax for internal defibrillation.

Past (and much current) practice regarding the placement of the electrodes for either type of defibrillator has been determined by clinical trial and error. One of our goals is

to allow engineers to use our computational steering model to assist in determining the optimum electrode placement, size, shape, and strength of shock to terminate fibrillation by solving equations (8.1-8.3) within a detailed model of the human thorax.

SCIRun uses computational steering to allow the user to interactively place the electrode(s) directly into the computer model of the thorax using a graphical input device and automatically change input parameters and boundary conditions as well as the mesh discretization level needed for an accurate finite element solution. The user then runs the simulation. As quickly as the machine allows, the user is presented a graphical representation of the effectiveness of the solution (via calculated current density in the visually represented heart, for example). The user then decides whether to continue the calculation on more refined meshes for higher accuracy, interactively change the position and/or design of the electrode, or both; or the user may allow the calculation to continue while designing another configuration. Although the example given above is targeted to a specific application in computational medicine, the tools created for such a problem have wide application in many engineering field problems and can be modified to other specific governing equations and geometries.

8.1.1 Geometric Modeling

In most computational engineering and science applications, a significant amount of geometric modeling must take place prior to simulation and visualization, as is the case in this application. Modeling efforts usually involve geometrical construction of a physical domain, in which a continuous structure must be discretized and adequately rendered into discrete spatial elements.

To solve the bioelectric field problems associated with equations (8.1-8.3), we have constructed torso models from MRI (magnetic resonance imaging) scans. A semiautomatic segmentation algorithm is used to classify the relevant tissues [109]. We then utilize 2D and 3D Delaunay triangulation algorithms to construct 3D surfaces and/or volumetric meshes [105, 127, 128]. Geometric postprocessing consists of assigning each tetrahedron an electrical conductivity tensor. The range of conductivity represented in the model is approximately 50 to 1 with blood the most conductive and bone the least. The largely parallel orientation of the skeletal muscle results in a higher conductivity along

the fibers than across the fiber direction, a fact that necessitated the use of conductivity tensors.

Construction of the geometric model is often one of the most time consuming aspects of the modeling and simulation process. For each new configuration, a new model must be assembled. Once a model has been constructed and simulations are running, a researcher must wait through an entire simulation before making changes to the geometry, or before learning if the changes already enacted have been effective. Since making such changes and recomputing the effects of those changes is time consuming, researchers are often restricted in the number of options they can effectively test.

In a computational steering framework, the goal is to change geometric features of the model or the spatial discretization of the solution domain in an interactive manner. Ideally, the user receives some degree of feedback on the calculation almost immediately and is allowed to change such input boundary conditions as, for example, spatial location and magnitude of a source, or the time step with which the calculation proceeds. These changes automatically trigger the computational and visualization phases of the problem. Such a framework allows more immediate access to simulation results and significantly reduces the time spent in making simulation and modeling design changes.

8.1.2 Numerical Analysis

Because of the geometric complexity of the solution domain (i.e., body) and anisotropic nature of some of the inhomogeneities (i.e., muscle) in our problem, the finite element (FE) method is the preferred choice to numerically approximate the solutions to equations (1-3). Application of the FE method yields a linear system, $\mathbf{A}\mathbf{\Phi} = \mathbf{b}$, where $\mathbf{A}$ is sparse, symmetric, and positive definite and on the order of hundreds of thousands to millions of degrees of freedom, depending on the level of mesh refinement, on the level of interactivity required by the user, and on the specific goals of the study [56, 58].

Due to the quasi-static nature of the bioelectric field problems we are solving [94], we are primarily interested in spatial steering for this application. Spatial steering, for example, involves controlling the discretization level of the geometry, typically in terms of the accuracy of the finite element model. We have implemented an adaptive method to automatically refine and de-refine a finite element mesh based upon a posteriori error es-

timates of the finite element approximation [132]. Briefly, suppose we want the accuracy of the finite element approximation to be within a given tolerance, δ of the true solution,

$$||\Phi - \Phi_h|| \leq \delta. \tag{8.4}$$

We can use an error estimator [23, 105, 55] to refine the mesh so that the resulting error in the finite element approximation of the governing equations decreases to within the limits imposed by equation (8.4). Such error estimators take the form, for example,

$$\sum_{k \in T_h} (h_k ||\Phi||_{H^2(k)})^2 \leq \frac{\delta^2}{c^2}, \tag{8.5}$$

where $||\Phi||_{H^2(k)}$ measures the L_2-norm of the partial derivatives of Φ, c is a positive constant, and h_k is a measure of the size of an element. The simulation starts with a coarse discretization level that conforms to the topology of the problem and a given set of boundary conditions. The error estimator is applied to all k elements that make up the finite element mesh, T_h. If the value of the error estimate in a given element exceeds the prescribed tolerance, then the element is refined according to the degree of the error. Obviously, with each iteration of the refinement algorithm, the number of degrees of freedom increases, as does the time necessary to compute subsequent iterations.

8.1.3 Scientific Visualization

Certainly, effective interpretation of computer simulations depends upon the visualization of the data. Traditionally, the visualization phase has been an entirely separate process from the computational phase. Computations are usually stored off to disk and/or piped into a separate visualization software package once all computations are completed. Furthermore, many scientists have relied on current “off the shelf” visualization packages that are not well suited for use with large engineering datasets (at least not in an interactive fashion). Within the computational steering modality, visualization is an integral part of the computational and geometrical modeling phases and is used to navigate the user through the data and/or to help the user interact with the data to modify the input parameters or geometric design. The user is able to visualize and explore intermediate results while the calculations continue to progress. Refined datasets are automatically substituted for the less accurate ones as they are completed.

8.1.4 SCIRun Implementation

A network that can be used to model cardiac defibrillation is shown in Figure 8.1. The network consists of the following modules:

- **SurfaceReader** reads a triangulated surface definition from a file. One of these modules reads the torso boundary (body surface) geometry, and the other reads the epicardium (heart surface) geometry.

- **GenSurface** generates two cylindrical electrodes for the defibrillation study. Parameters in the interface allow the scientist to control the shape, applied voltages, and discretization of the electrodes. GenSurface also tags the surface with appropriate boundary condition flags for the problem. The torso and scalp have a zero-flux Neumann boundary. In the defibrillation problem, the two electrode cylinders have Dirichlet boundary conditions corresponding to their respective voltage. The voltage sources may be changed interactively.

- **VisualizeMatrix** draws a figure representing the nonzeros in a matrix. Non-zero entries are draw with small red dots, and zero entries are left black. This gives the user a quick representation of the sparse structure of the matrix. A magnifying glass can reveal the actual numbers in a portion of the matrix by clicking the mouse in the desired region. Although this is not a necessary part of the simulation, it is a handy tool for learning about or debugging the finite element matrix. Using it simply requires attaching the VisualizeMatrix module to the appropriate matrix output.

- **SurfToGeom** converts the surface definitions into displayable geometry. A toggle button in the user interface controls whether or not the geometry is movable. The epicardium and torso boundary should not be moved, since they correspond to physical geometry. The electrode cylinders, on the other hand, must be moved so that various placements can be tested. The SurfToGeom module provides 3D widget handles that allow the user to manipulate these surfaces directly. An optional input parameter selection maps scalar field values onto the surface. This input is attached

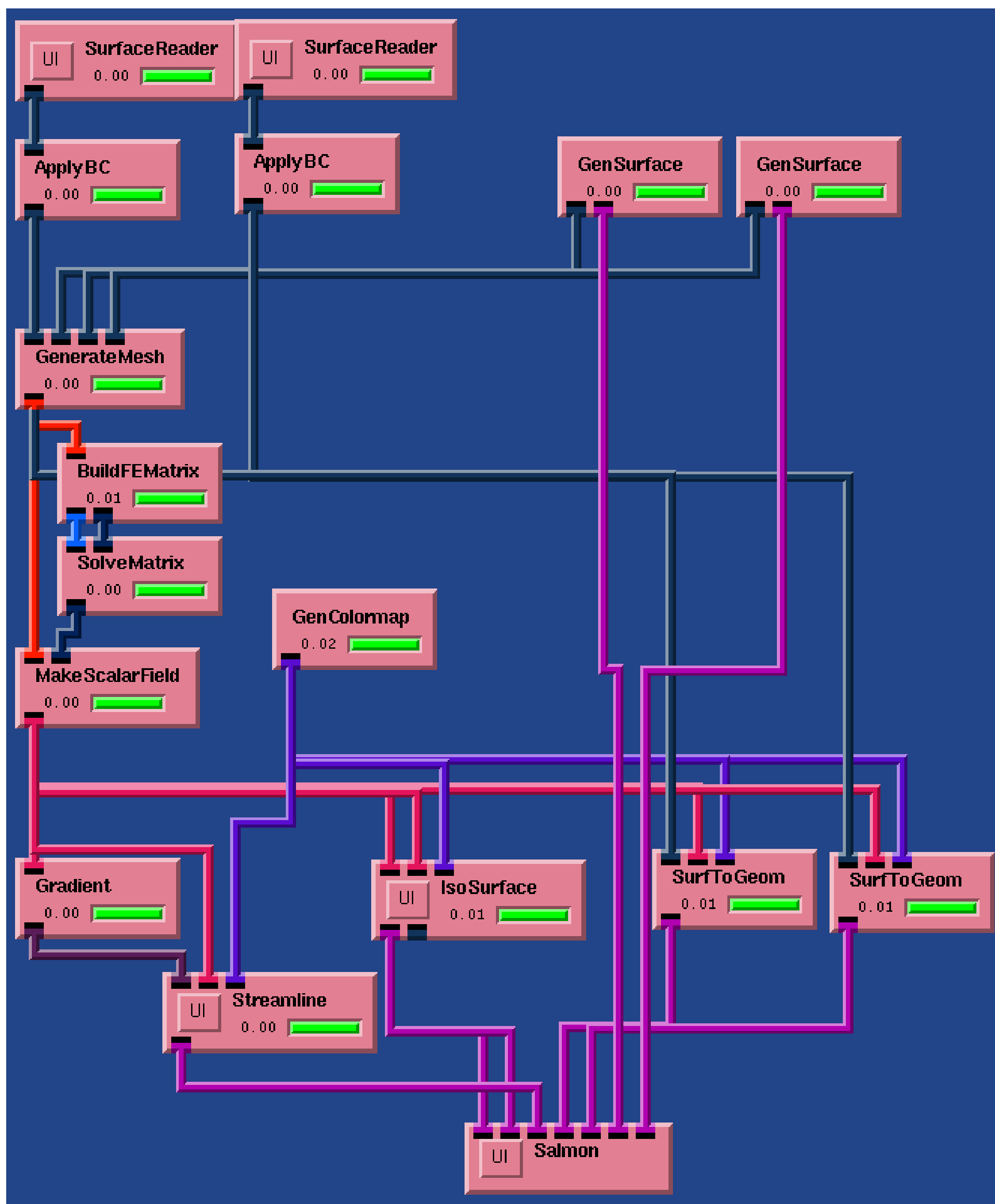

Figure 8.1. An example of a fairly complex dataflow network, showing the modules (the boxes), the connections (the wires between them), and the input/output ports (the points on the modules that the wires connect). On a color monitor, the colors of the ports and connections indicate the type of data that flows through them.

to the epicardium and shows the voltages on the surface of the heart.

- **MeshReader** reads a mesh from a file that contains the discretized torso geometry. The user can read different meshes that contain different anatomical features, different discretization levels, etc.

- **InsertDelaunay** inserts the nodes creates by the defibrillator sources into the mesh read by MeshReader. It uses a Watson [127] algorithm for inserting nodes into a Delaunay tetrahedral mesh.

- **BuildFEMatrix** applies the finite element method to the governing equations, using the mesh structure and the boundary conditions to construct a matrix that describes the user-specified configuration. Utilizing controls on the user interface, the user may instruct the module to create a dense matrix, a band diagonal matrix or a compressed sparse-row matrix.

- **SolveMatrix** uses direct or iterative algorithms to find the solution to the matrix equation. For this problem, we use a preconditioned conjugate gradient algorithm for iterative solutions. The scientist controls convergence parameters and algorithm selection through the graphical user interface.

- **MakeScalarField** combines the solution of the finite element matrix to the volume mesh generated by GenerateMesh. This mesh/solution combination provides a representation of the solution in terms of a scalar field of voltage values.

- **IsoSurface** allows interactive extraction of Iso-surfaces in the voltage field. A small sphere controls the starting point of the isosurface algorithm, and an attached 3D arrow shows the direction of the gradient. The sphere and arrow widget may be moved using the mouse to allow interactive exploration of the voltage field. Dragging on the body of the arrow moves the widget along the line defined by the gradient; dragging on the sphere allows unconstrained movement of the seed point. Alternatively, the user can select a voltage directly through a 2D slider.

- **Gradient** computes a vector field from the scalar voltage field according to equation (8.3). This yields another form of the solution in terms of electric current density.

- **Streamline** produces vector field lines that reveal the flow of electrical currents within the torso. These field lines are analogous to massless particle traces in fluid flow fields. The streamlines are advected using a fourth-order Runge-Kutta technique. The user may choose between a single streamline or a row of streamlines. Time-step adaptation parameters and step sizes are controlled via the 2D user interface, while the positions of the particle sources are controlled with 3D widgets [44].

- **CuttingPlane** interpolates a planar slice through the unstructured data and maps data values to colors on a semitransparent surface. The plane can be manipulated with a 3D widget to allow the user to look at different cross sections of the electric potential.

- **GenColorMap** generates a colormap which is used to map voltage values to colors. The output of this module is used by the SurfToGeom, Streamline, IsoSurface, and CuttingPlane modules. The colormap can be changed interactively.

- **Salmon** provides the underlying structure for viewing geometry and 3D user interaction for both viewpoint control and control of the 3D widgets described above. As the Streamline module computes streamlines, or as the Isosurface module computes isosurfaces, it sends geometry (lines, triangles or other primitives) to Salmon. Salmon displays all of these objects geometry in an interactive rendering window.

Each of these modules is simple enough to be managed easily, but when they are joined together, they accomplish a complex task. A sample visualization from this type of network is shown in Figure 8.2.

The user can select a new electrode configuration by a moving a 3D widget in the salmon window. When the user releases the mouse button, SCIRun reexecutes the dataflow program. The finite element mesh is regenerated automatically, and a finite

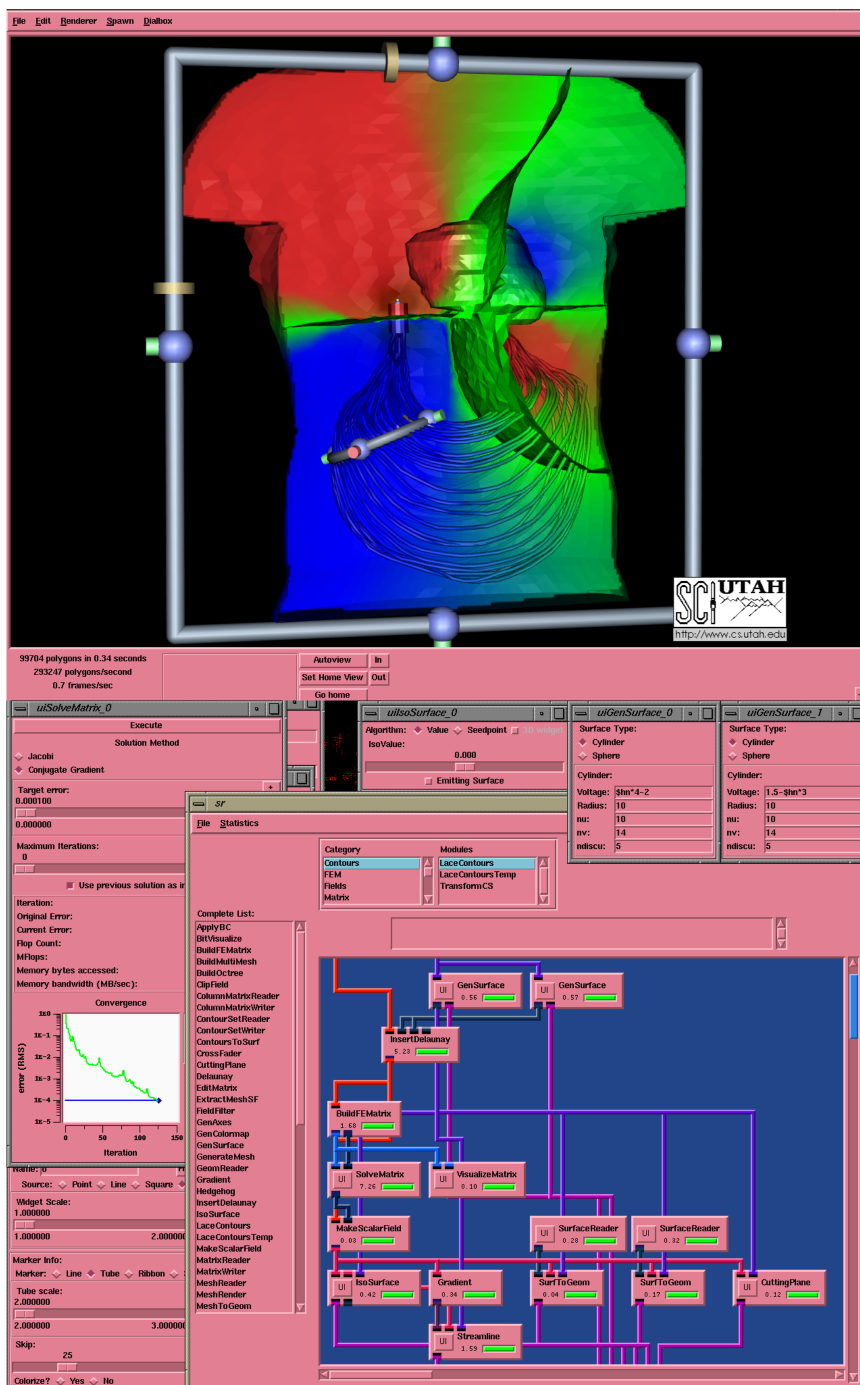

Figure 8.2. Visualization of a defibrillator design simulation, showing an electrode, the surface of the heart (epicardium), a 3D widget (circular rake), and electrical current lines (streamlines). The other electrode in the simulation is obscured by the heart. Much of the current leaving the visible electrode travels away from the heart, due to the high conductivity of blood in a nearby vessel.

element matrix is created that reflects the modifications to the electrode configuration. The matrix solver and proceeds to iteratively seek a solution to the system of equations. As the matrix solver computes, partially converged results are sent to the visualization modules. For a long-running simulation (several minutes to several hours), this can provide valuable insight to the effectiveness of a solution long before the solver has fully converged.

8.2 Application: Monte Carlo Global Illumination

The rendering equation [61] is a Fredholm integral equation of the second kind that describes the transport of light in an environment:

$$I(x, x') = g(x, x')[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''], \tag{8.6}$$

where $I(x, x')$ is the intensity of light passing from x' in the direction of x, $\epsilon(x, x')$ is the emitted light intensity from x' in the direction of x, $\rho(x, x', x'')$ is the intensity of light reflected from x'' to x at the point x', S is a collection of all surfaces in the scene, and $g(x, x')$ is a binary function with value 0 if x' is not visible from x, and 1 otherwise. Using the rendering equation, one can create a physically realistic model of the reflection of light in a scene. Monte Carlo methods are a popular method [110] for solving this equation directly. These simulations are typically long-running, CPU-intensive computations [38].

The Monte Carlo sampling is typically performed by evaluating the light spectrum that enters a virtual camera. Since the computation is done in spectral units, it must be transformed into RGB colors suitable for display [41]. To complicate matters, the resulting RGB colors are not always within the range displayable on a computer monitor. There are a wide variety of methods used to compress the gamut of the image, but they vary in subjective quality from image to image [93].

Computational steering is used in the example to allow the user to control the spectrum to RGB transformation pipeline even as the Monte Carlo sampling is still occurring. This allows a researcher to examine early results of the rendering using different transformation functions.

8.2.1 SCIRun Implementation

The SCIRun network used to implement this is shown in Figure 8.3, and an image output from this is shown in Figure 8.4. The image is a model of the "Cornell Box", a benchmark scene that researchers at Cornell have used to compare the results of their renderings to images of the real model [25]. SCIRun is used to set up the scene and to tune various parameters. Figure 8.5 shows this simulation as constructed in the SCIRun environment. These runs take from a few seconds during the initial coarse adjustments to several hours for near-final tuning. Once the parameters are set, a longer running simulation is performed to compute a high quality image. The final image is 2048 by 2048 pixels, with 100k (317^2) samples at each pixel, and traces up to 25 paths for each sample. This image took over 2 CPU years to execute on 195 Mhz R10000 processors. Using checkpointing, it was computed on varying numbers of processors over the course of about 2 months.

This is an example of the fine-grained dataflow setup described in Chapter 4 [114].

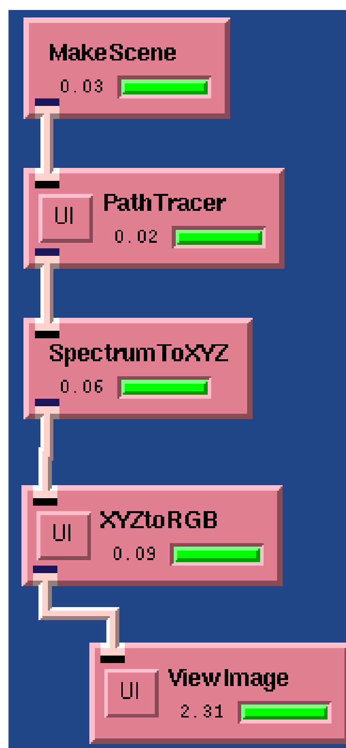

Figure 8.3. The dataflow program for a path tracing program. The imaging pipeline can be manipulated even while the program executes.

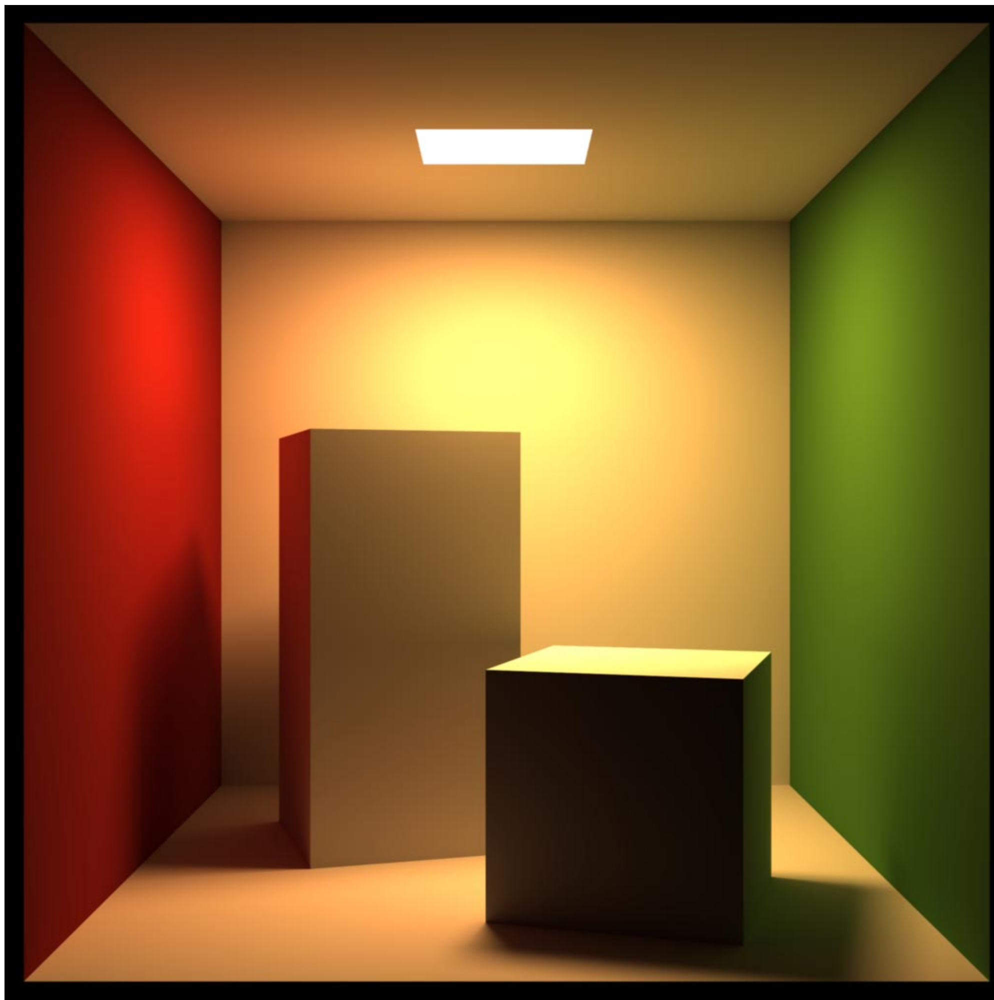

Figure 8.4. The Cornell Box as generated by the SCIRun PathTracer Module and imaging pipeline
.

In a coarse-grained dataflow model, the entire dataset (the spectrum image) would be transferred when the PathTracer module completes. This does not allow the user to interact with intermediate results. The fine-grained dataflow model here allows pixels to be sent down the pipe independently. For the sake of efficiency, a shorter running program might choose to send larger chunks of the image, up to the full image at once.

The following modules are used to implement this application:

- **MakeScene** produces a scene, which includes a camera position, light sources, and objects in the worlds to be rendered. The objects are represented as a scene graph.

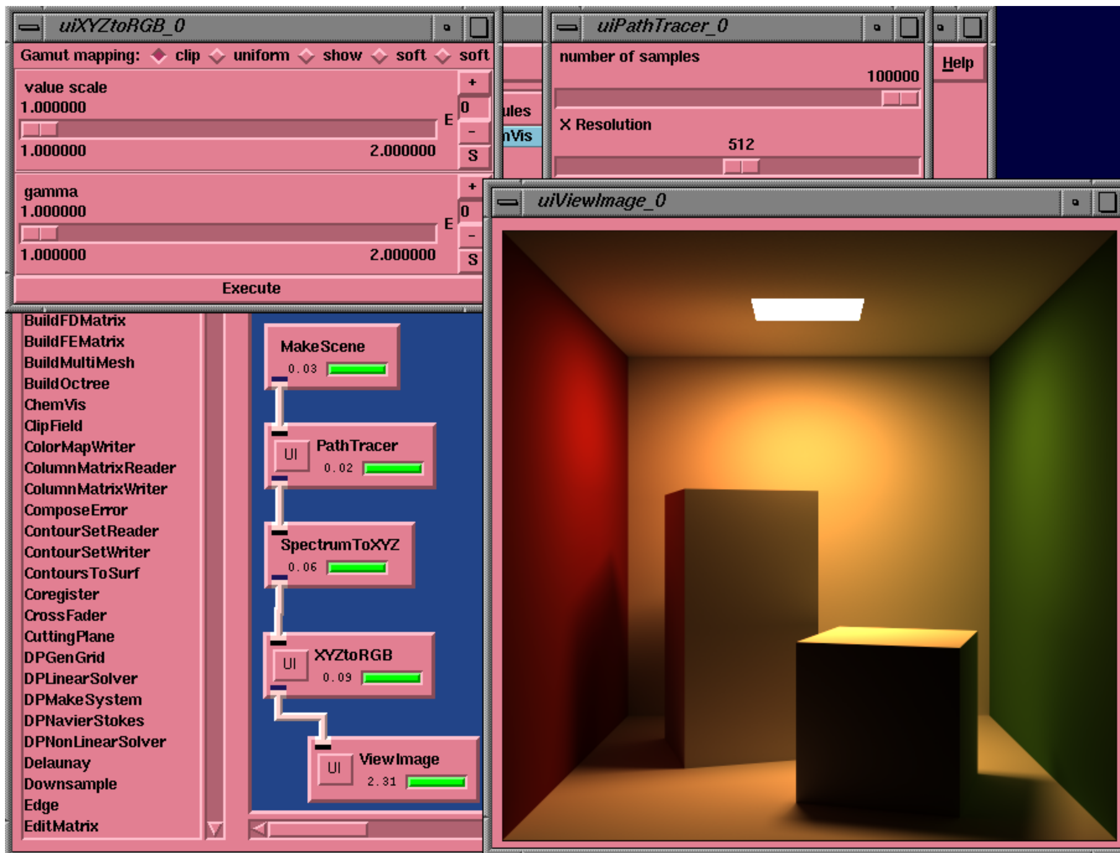

Figure 8.5. An example of the SCIRun path tracer, showing the interface to steer the adaptation parameters while the program is still executing.

- **PathTracer** performs the actual Monte Carlo integration of the rendering equation for a particular scene. Produces an image that contains a sampling of a spectrum at each pixel.

- **SpectrumToXYZ** transforms the spectrum at each pixel to the XYZ color space [41].

- **XYZtoRGB** transforms each pixel from the XYZ color space to RGB colors, ready for display. The user interface is used to select the gamut compression algorithm used.

In addition to the controls on the color space transformation modules, the user can directly manipulate the sampling density and adaptation criteria as the program is running. This allows the user to trade off image quality and execution time based on their

examination of the partial images.

8.3 Application: CFD Applications Using CFDLIB

CFDLIB [4] is a library of computer codes written at Los Alamos National Laboratory. It solves a variety of Computational Fluid Dynamics (CFD) problems in 2D and 3D. CFDLIB is written entirely in Fortran 77. It offers only the simulation component of the problem; scientists typically use third party visualization and modeling packages or sometimes perform modeling manually. It is the epitome of the batch-oriented, non-interactive scientific simulation program described in Chapter 1.

CFDLIB differs from other applications discussed above, in several major points:

- It produces a time-varying field, as opposed to a static field. This is accommodated in the dataflow model, but a more specific treatment of time-based data is left for future work.
- It is an existing program that must be “shoe-horned” into the SCIRun environment. This process includes instrumenting the code to extract data at appropriate points. It also includes modifying the code to enable steering of specific parameters during program execution.
- It is written in Fortran instead of C++. Although not a research issue, this posed a significant challenge when integrating it within the object-oriented SCIRun environment. Wrapper functions were written in C that allow C++ objects to be manipulated with Fortran subroutine calls.
- The internal data structures differ from SCIRun’s native field formats.

In this example, SCIRun is used to extract the pressure, temperature and velocity fields from the running simulation as well as the particle sets resulting from a particle-based simulation.

8.3.1 SCIRun Implementation

In this example, the entire CFDLIB package is imported as a single module within SCIRun. Several changes were made to the CFDLIB program:

1. The main program of the Fortran code is changed into a subroutine.

2. A C++ module is created as a wrapper to the legacy program.

3. A subroutine that wrote out data files is changed to make calls to the SCIRun Fortran wrappers. Instead of writing data to disk, it sends data through the SCIRun dataflow graph.

An example of this network is shown in Figure 8.6 and the resulting visualization is shown in Figure 8.7. The CFDLIB module creates a thread that runs the Fortran

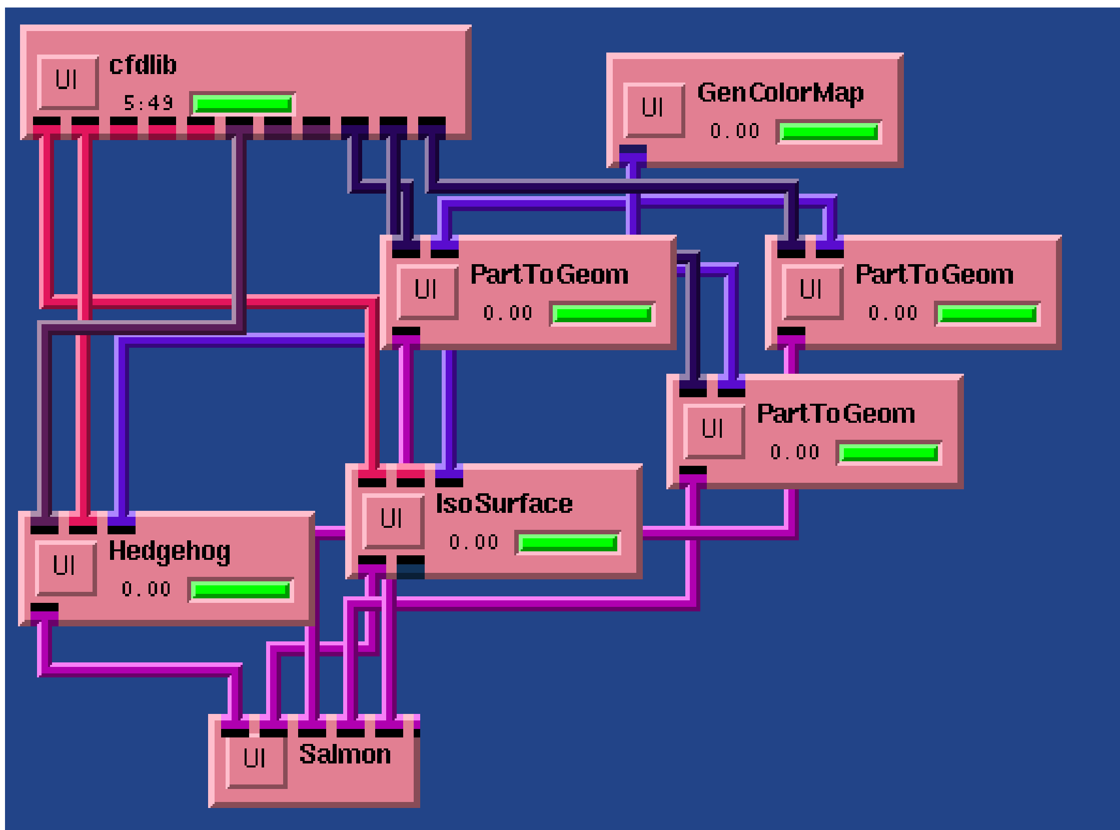

Figure 8.6. The dataflow program for an example Fortran program (CFDLIB). The CFDLIB module contains the entire Fortran simulation, and various visualization and postprocess modules are connected to it.

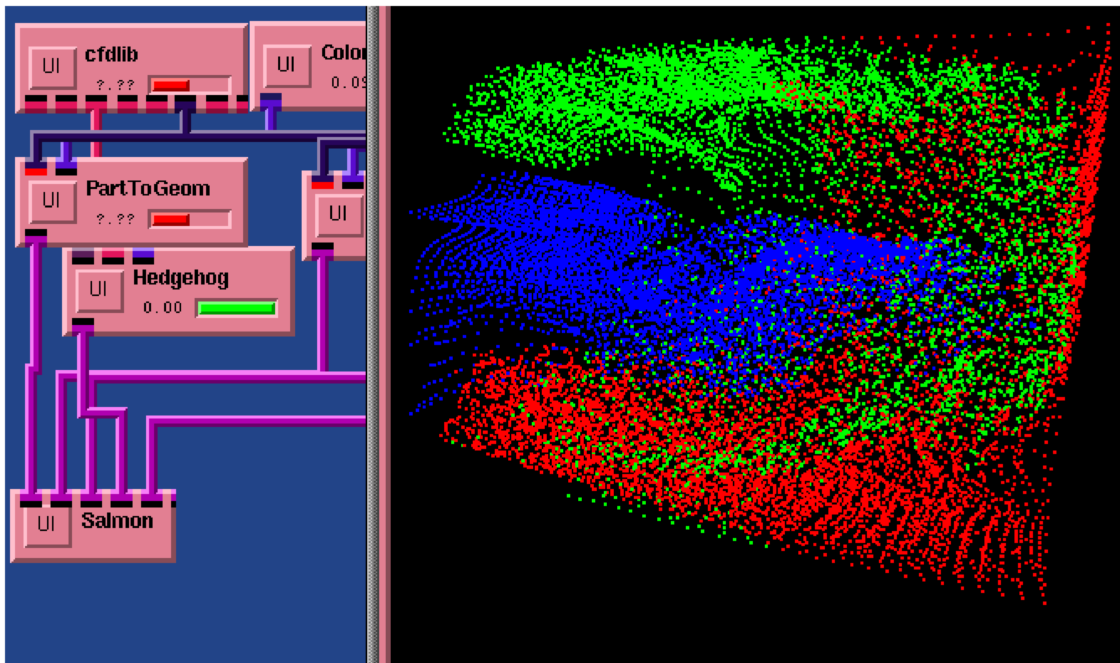

Figure 8.7. The dataflow program and resulting visualization for an example Fortran program (CFDLIB). The simulation consists of three fluids of different density that interact inside of a box.

code. Callbacks inserted into the CFDLIB code update the time-varying datasets in the CFDLIB module. A slider in the module interface allows the user to view datasets from previous time steps.

The following modules were used in this simulation:

- **cfdlib** is he adapter module which calls the CFDLIB fortran program.
- **PartToGeom** converts particle sets produced by the simulation to renderable geometry. The particles can be displayed as points or as small spheres. There are several of these modules - one for the particles associated with each fluid.
- **GenColorMap** produces a colormap that maps a fluid parameter to a color in the visualization. This colormap is
- **Hedgehog** produces arrows at sample points in a vector field. The length of the arrow indicates the magnitude of the field at that point, and the orientation indicates the direction of the field. In addition, the shaft of the arrow is mapped to a scalar

field value using a colormap produced by GenColormap.

- **IsoSurface** allows interactive extraction of Iso-surfaces in a scalar field.

- **Salmon** provides an interactive view of the simulation results. The geometric objects produced by PartToGeom, Hedgehog, and IsoSurface are displayed interactively. The user can turn particular objects on and off, and can interact with 3D widgets produced by the Hedgehog module.

In this simulation, scalar fields of density, temperature, and volume fraction are produced for each type of fluid. In addition, a single field of pressure is produced for all fluids. A vector field of velocity is also generated.

The GenColorMap, IsoSurface, and Salmon Modules were identical to those used in the torso defribrillator application described in Section 8.1.

8.3.2 Fortran Code

Several difficulties were encountered when trying to integrate Fortran programs in SCIRun. These were as follows:

1. **Problem:** Fortran I/O.

   Consequence: Fortran modules that perform I/O append all of their output from multiple runs in a single file. This can be worked around by explicitly closing the Fortran files at the end of execution.

2. **Problem:** Common blocks.

   Consequence: Only one instance of a Fortran module can be executing at any time. It can also have a potential conflict between different Fortran modules, especially if unnamed common blocks are used. This problem could be overcome through mangling of symbol names, but SCIRun does not presently attemp this.

3. **Problem:** Locally declared arrays.

   Consequence: Locally declared arrays in Fortran can cause the stack to grow by large amounts. This can easily overflow the small stacks that the SCIRun thread

library provides by default. The module must request a large stack, or the stack will overflow.

4. **Problem:** Save statements and other global data.

   Consequence: Simulations may not function correctly if executed more than once in a single SCIRun session. Even though global data in any language can cause the same problem, it seems to be more pervasive in many Fortran programs. The solution is to provide functions which can reset the data at the beginning of a simulation.

8.3.3 Results

In this example, the simulation and modeling components consist of only a single module. However, this module can be connected to a wide variety of visualization and analysis tools. In other examples where legacy code is integrated, a simulation module might be adapted to employ the SCIRun matrix solvers, mesh generators, or other algorithms.

The utility of the CFDLIB/SCIRun combination can be extended beyond visualization. By adding a few lines of code to CFDLIB, one can turn different simulation variables into user steerable parameters. An example of this is the time step parameter. It is simple to add this statement:

```
call cfdlib_get_timestep(module, dt)
```

at the beginning of the time integration loop. This will call the cfdlib object to retrieve the new timestep and store it in the dt variable. This allows the time step parameter to be controlled through a user interface slider. By contrast, the traditional way to change this parameter is to stop the simulation, modify the “in3d” input deck, and start the simulation again from the beginning. Although it may not be wise to change the timestep radically on a whim, it is a natural parameter to experiment with when developing new simulations.

These modifications allow any of the CFDLIB simulations to be executed within SCIRun. The simulations run within a few percent of the original speed of the standalone Fortran program. We have found the greatest utility executing simulations that take tens

of minutes to a few hours. In Figure 8.7, the simulation consisted of 20,000 particles and executed 200 time steps (adaptive time stepping) in just under 6 minutes an an SGI Octane with a 225 Mhz R10000 processor.

8.4 Application: Atmospheric Dispersion

This application is the product of a collaboration between the Scientific Computing and Imaging research group and the Martin Berzins at the University of Leeds. We worked to incorporate his model of atmosopheric dispersion as a SCIRun module, and were then able to combine it with the visualization and unstructured mesh generation components in SCIRun.

Following the discussion as reported in [52], the application we consider here is taken from a model of atmospheric dispersion from a power station plume – a concentrated source of NO_x emissions [10, 43]. The photo-chemical reaction of this NO_x with polluted air leads to the generation of ozone at large distances downwind from the source. An accurate description of the distribution of pollutant concentrations is needed over large spatial regions to compare with field measurement calculations. The present trend is to use models incorporating an ever larger number of reactions and chemical species in the atmospheric chemistry model. The complex chemical kinetics in the atmospheric model gives rise to abrupt and sudden changes in both space and time in the concentration of the chemical species in both space and time. These changes must be matched by changes in the spatial mesh and the timesteps if high resolution is required [119]. The difference in time-scale between the reaction of these species leads to stiff systems of equations that require implicit numerical solvers and special linear equations solvers [10]. The requirements of such a problem are that it is necessary to combine:

- Unstructured tetrahedral mesh generation and adaptation.
- Physically realistic spatial discretization methods.
- Stiff ODE integrators tailored to the application.
- Fast interactive visualization for multispecies flows

- Computational steering facilities for transient problems.

These requirements were met by combining SCIRun with the spatial discretization, mesh adaptation and time integration codes CSPRINT and TETRAD [10, 115] and by wrapping these codes in SCIRun modules, with converters to map to the SCIRun data structures.

Following the development of the model as reported in [52], the power plant plume application is modeled by the atmospheric diffusion equation in three space dimensions given by:

$$\frac{\partial c_s}{\partial t} = -\frac{\partial u c_s}{\partial x} - \frac{\partial v c_s}{\partial y} - \frac{\partial w c_s}{\partial z} + D(\frac{\partial c_s}{\partial x}, \frac{\partial c_s}{\partial y}, \frac{\partial c_s}{\partial z}) + R_s(c_1, c_2, ..., c_q) + E_s - (\kappa_{1s} + \kappa_{2s})c_s, \tag{8.7}$$

where c_s is the concentration of the s'th compound, u,v and w are wind velocities, K_x and K_y are diffusivity coefficients, and k_{1s} and k_{2s} are dry and wet deposition velocities respectively. E_s describes the distribution of emission sources for the sth compound and R_s is the chemical reaction term that may contain nonlinear terms in c_s. $D()$ is the diffusion term, which is set to zero here. For n chemical species an n-dimensional set of partial differential equations (PDE's) is formed where each is coupled through the nonlinear chemical reaction terms.

The test case model covers a region of 300 x 500 km and is a 3D form of that used by [43], and although far from as detailed, it does represent the main features that would commonly be found in an atmospheric model including slow and fast nonlinear chemistry, namely, concentrated source terms and advection The chemical mechanism contains only seven species but still represents the main features of a tropospheric mechanism, namely the competition of the fast inorganic reactions $NO_2 \overset{O_2}{\rightarrow} O_3 + NO$ and $NO + O_3 \rightarrow NO_2 + O_2$ with the chemistry of volatile organic compounds (VOCs), which occurs on a much slower time-scale. This separation in time-scales generates stiffness in the resulting equations. The reaction rate constants have been chosen as in Tomlin et al. [119], and the photolysis rates were parametrized as a function of the solar zenith angle(see [119]). The background concentrations listed by [43] form the initial conditions for the model. These concentrations will then change diurnally as the

chemical transformations take place. The power station is taken to be the only source of NO_x and this source is treated by setting the concentration in the chimney set as an internal boundary condition. In terms of the mesh generation this ensures that the initial grid will contain more elements close to the concentrated emission source. The concentration in the chimney corresponds to an emission rate of NO_x of 400kghr^{-1} and only 10% of the NO_x to be emitted as NO_2. We have assumed a constant wind speed of 5ms^{-1} in the x-direction with y and z components of one tenth of this value.

8.4.1 Spatial Discretization Method

Spatial discretization of the model atmospheric diffusion equation on unstructured tetrahedral meshes reduces the set of PDEs in four independent variables to a system of ordinary differential equations (ODEs) in one independent variable: time. This system of ODEs can then be solved as an initial value problem, using the software tools that exist for this purpose [10]. For advection dominated problems it is important to choose a discretization scheme that preserves the physical range of the solution [115]. The method used here is a cell-centered, finite volume discretization scheme of [115] that enables accurate solutions to be determined for both smooth and discontinuous flows by making use of the upwind techniques for the advective parts of the fluxes.

8.4.2 Time Integration

The time integration method used is the theta method module of the CSPRINT software, which is designed for the moderate accuracy solution of stiff systems using local error control in time [10]. Once the PDEs have been discretized in space we are left with a large system of coupled ODEs of dimension m $\times$ n where m is the number of mesh points and n the number of species. These equations may now be written for a single species as

$$\dot{\underline{U}} \ = \ \underline{F}_N \ \ (\, t, \, \underline{U}(t) \,) \ \ , \ \underline{U}(0) \ \text{ given} \ , \tag{8.8}$$

where $\underline{U}(t) \ = \ [U(x_1, y_1, z_1, t), ..., U(x_N, y_N, z_N, t)\ ]^T$. The point x_i, y_i, z_i is the centroid of the i th cell and $U_i(t)$ is a numerical approximation to the exact solution to the PDE evaluated at the centroid, i.e., $u(x_i, y_i, z_i, t)$. The time integrator computes an approximation, $\underline{V}(t)$, to the vector of exact PDE solution values at the mesh points. This

numerical solution at $t_{n+1} = t_n + k$, where k is the time step size, as denoted by $\underline{V}(t_{n+1})$, is computed from

$$\underline{V}(t_{n+1}) \;=\; \underline{V}(t_n) \;+\; (1-\theta)k\,\dot{\underline{V}}(t_n) + \theta\, k\, \underline{F}_N(t_{n+1}, \underline{V}(t_{n+1})), \tag{8.9}$$

in which $\underline{V}(t_n)$ and $\dot{\underline{V}}(t_n)$ are the numerical solution and its time derivative at the previous time t_n and $\theta = 0.55$. The equations to be solved for the correction to the solution $\underline{\Delta V}$ for the $p+1$ th iteration of the modified Newton iteration used with the Theta method are:

$$[I - k\theta J]\,\underline{\Delta V} \;=\; \underline{r}\,(t^p_{n+1}) \tag{8.10}$$

where $J = \frac{\partial\, \underline{F}_N}{\partial \underline{U}}$, $\underline{\Delta V} \;=\; \left[\underline{V}(t^{p+1}_{n+1}) - \underline{V}(t^p_{n+1})\right]$ and

$$\underline{r}\,(t^p_{n+1}) \;=\; -\,\underline{V}(t^p_{n+1}) + \underline{V}(t_n) + (1-\theta)k\dot{\underline{V}}(t_n) - \theta k \underline{F}_N(t_{n+1}, \underline{V}(t^p_{n+1})). \tag{8.11}$$

The solution of this system of equations constitutes the major computational task of the calculation. The CPU times are excessive unless special solution techniques such as splitting the nonlinear equations [10] into a set of flow terms and a reactive source term are employed. Consider the ODE function $\underline{F}_N\;(t,\; \underline{U}(t)\;)$ defined by equation (5.2) and decompose it into two parts:

$$\underline{F}_N\;(t,\; \underline{U}(t)\;) \;=\; \underline{F}^f_N\;(t,\; \underline{U}(t)\;) \;+\; \underline{F}^s_N\;(t,\; \underline{U}(t)\;) \tag{8.12}$$

where $\underline{F}^f_N\;(t,\; \underline{U}(t)\;)$ represents the discretization of the convective flux terms f and g in equation (1) and $\underline{F}^s_N\;(t,\; \underline{U}(t)\;)$ represents the discretization of the of the source term h in the same equation. The splitting approach, used in [10], is to employ the following approximation to the Jacobian matrix used by the Theta method within a Newton iteration:

$$I - k\theta J \;\approx\; [I - k\theta\; J_f] \quad [I - k\theta\; J_s\,]\;) \;+\; O(k^2). \tag{8.13}$$

where $J_f \;=\; \frac{\partial\, \underline{F}^f_N}{\partial \underline{U}},\;\; J_s \;=\; \frac{\partial \underline{F}^s_N}{\partial \underline{U}}$. The new iteration may thus be written as

$$[I - k\theta J_s]\;\; \underline{\Delta V^*} \;=\; \underline{r}\,(t^p_{n+1}) \tag{8.14}$$

where $\underline{\Delta V^*}$ is the operator splitting approximation to $\underline{\Delta V}$. The advantage of this is that each block of equations corresponding to a tetrahedral element may be solved separately

using the Gauss Seidel method of Verwer [122]. The Jacobian matrix $[I - k\gamma J_s]$ is split into L, the strictly lower triangular, D, the diagonal, and U, the strictly upper triangular matrices. The equation is rearranged to get

$$(I - \gamma kD - \gamma kL)\Delta \underline{V}^*_{m+1} = \gamma kU \Delta \underline{V}^*_m + \underline{r}(t^p_{n+1}). \tag{8.15}$$

This approximation introduces a splitting error that fortunately only alters the rate of convergence of the iteration as the residual being reduced is still that of the full ODE system. A lack of convergence of this iteration is dealt with by reducing the timestep k. The matrix $I - k\theta J_s$ is the Jacobian of the discretization of the time derivatives and the chemistry source terms. This matrix is thus composed of independent diagonal blocks with as many block as there are tetrahedra. Each block has as many rows and columns as there are PDEs. and each block's equations may be solved independently . The choice of a time step is a difficult issue in reacting flow problems; however in this case the chemistry reacts quickly compared to wind speed. The approach here is thus to use a standard local error control, though it is often the case that the convergence of the iteration that limits the timestep.

8.4.3 Mesh Generation and Adaptation

The initial unstructured meshes used are created from a geometry description using the SCIRun [92] mesh generator. The initial mesh inside a rectangular bounding box is generated with approximately 5000 elements. This resulted in a largest element with a side length 50 km. It is difficult to directly relate the size of unstructured meshes to regular rectangular ones, but our original mesh is comparable to the size of mesh generally used in regional scale atmospheric models. The fine scale grids used in present regional scale models are of the order of 10-20 km. For a power plant plume with a width of approximately 20 km, it is impossible to resolve the fine structure within the plume using grids of this size [119], hence our use of adaptive grids. Close to the chimney the mesh is refined to elements of length 5 km or 500 m depending on the mesh level used. This ensured that the mesh would be refined to a reasonable resolution in this region of steep gradients.

These meshes are then refined and coarsened by the TETRAD [115] mesh adaptation module that is based on the refinement of tetrahedra into eight tetrahedra with appropriate adjustments to ensure that the mesh is conforming at the edges. The criterion for the application of the adaptivity used in this work is based on refining or derefining the mesh based on the magnitude of solution gradients of the key chemical species NO and NO_2 across the faces of the tetrahedron [115]. For applications such as atmospheric modeling it is important that a maximum level of refinement can be set, to prevent the code from adapting to too high a level in regions with concentrated emissions. This is especially important if sources that are close to point sources in nature exist. For the test problem here the maximum level of refinement is a user-defined parameter that is often limited to level 2 or 3. At the same time a sufficiently small refinement tolerance must be set in order to ensure enough refinement to determine the detail of the plume.

8.4.4 Integration with SCIRun

The integration of the above routines with SCIRun required writing a few bridging functions that can be called both from SCIRun and the program to provide control parameters for the program and get the feedback during the execution. No other changes were made to SCIRun. The transient aspect of the problem is dealt with by the integration module TETRAD/SPRINT sending out a new mesh on every time step (rarely) or more usually immediately after to each remesh. While the module continues time integration, the rest of SCIRun network would process and visualize the current mesh.

The major advantage of SCIRun is provided by computational steering, i.e., the possibility not only to set up initial conditions and parameters but also to have control over the execution. In the case of TETRAD, the user interface allows the user to set up initial parameters of the problem: the position of the pollution source, initial velocity of the flow, and level of refinement. If, in the process of execution, the user decides that the refinement level or the refinement tolerance is to high or too low, then it can be changed for the next refinement. Similarly the species used as the basis for refinement can be altered dynamically, without quitting and losing any part of the data. An important ”What-if?” question is to ask about the effect of the changes in the wind velocity on the

existing solution. Accordingly we allowed the user to change the wind velocity during the execution. At the same time the visualization module provided 3D visualization of the flow, so without stopping computation one can examine the plume from different directions and angles and explore its cross section as the plume develops.

This example shows that SCIRun not only can be used with its “native” modules but can also successfully play a role of a framework to support all different kind of scientific computations.

8.4.5 Atmospheric Diffusion Simulation Results

As reported in [52], each run is carried out over a simulation period of 48 hours so that the diurnal variations can be observed. We present here only a selection of the results that illustrate the main features relating to the adaptivity and to the use of SCIRun. The main area of mesh refinement is along the plume edges close to the chimney, indicating that there is a high level of structure in the plume. Using the adaptive mesh, we can clearly see the plume edges and can easily identify areas of high concentrations. The effects of the plume on ozone concentrations also provide some interesting results. Close to the plume the concentration of O_3 is much lower than that in the background. Due to the high NO_x concentrations the inorganic chemistry is dominant in this region and the ozone is consumed by the second reaction in Section 8.4. As the plume travels downwind and the NO_x levels decrease, the plume gradually picks up emissions of VOCs, as shown in Figure 8.8. The OVC chemistry leads to the production of NO_2 that pushes the above reaction in the reverse route. The levels of ozone can therefore rise above the background levels at quite large distances downwind from the source of NO_x [52]. The following SCIRun modules were used to create this simulation:

- **MeshReader** reads a previously generated mesh from a SCIRun persistent object file.
- **GenTransferFunc** is similar to the GenColormap module described in the torso difibrillator application but provides opacities in addition to colors.
- **Advect8** provides a wrapper to the TETRAD program.

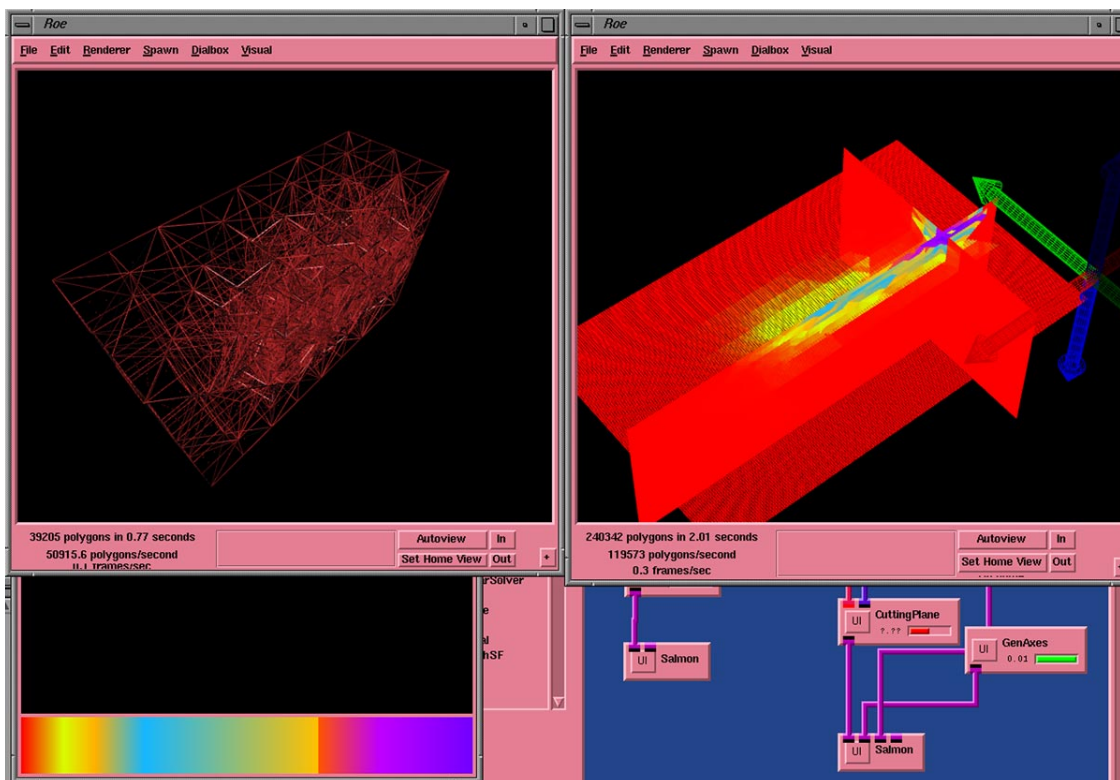

Figure 8.8. This picture shows one component of the plume in greater detail in three perpendicular cross sections.

- **RescaleColormap** finds the minimum and maximum values in a scalar field and scales the range of the colormap accordingly.
- **GenAxes** creates an icon in the scene, which displays arrows for the X, Y, and Z euclidian directions.
- **CuttingPlane** interpolates a planar slice through the unstructured data and maps data values to colors on a surface.
- **MeshToGeom** creates a representation of the mesh with lines or cylinders. The mesh can subsequently be rendered in the same view as the other visualizations of the simulation.
- **Salmon** provides an interactive view of the visualization tools. Multiple views can be used to display different variables, as in Figure 8.8.

Figure 8.9 shows the SCIRun network which implements this application.

The RescaleColorMap, CuttingPlane, and Salmon modules were identical to those in the previous simulations in this chapter.

8.5 Discussion

The four applications that we have just demonstrated show how SCIRun can be used in a wide variety of applications to steer computational applications. The torso defibrillator example demonstrated the power of SCIRun's computational components. The Monte Carlo global illumination model demonstrated SCIRun's imaging pipeline and how SCIRun combines coarse-grained dataflow with fine-grained dataflow. Third,

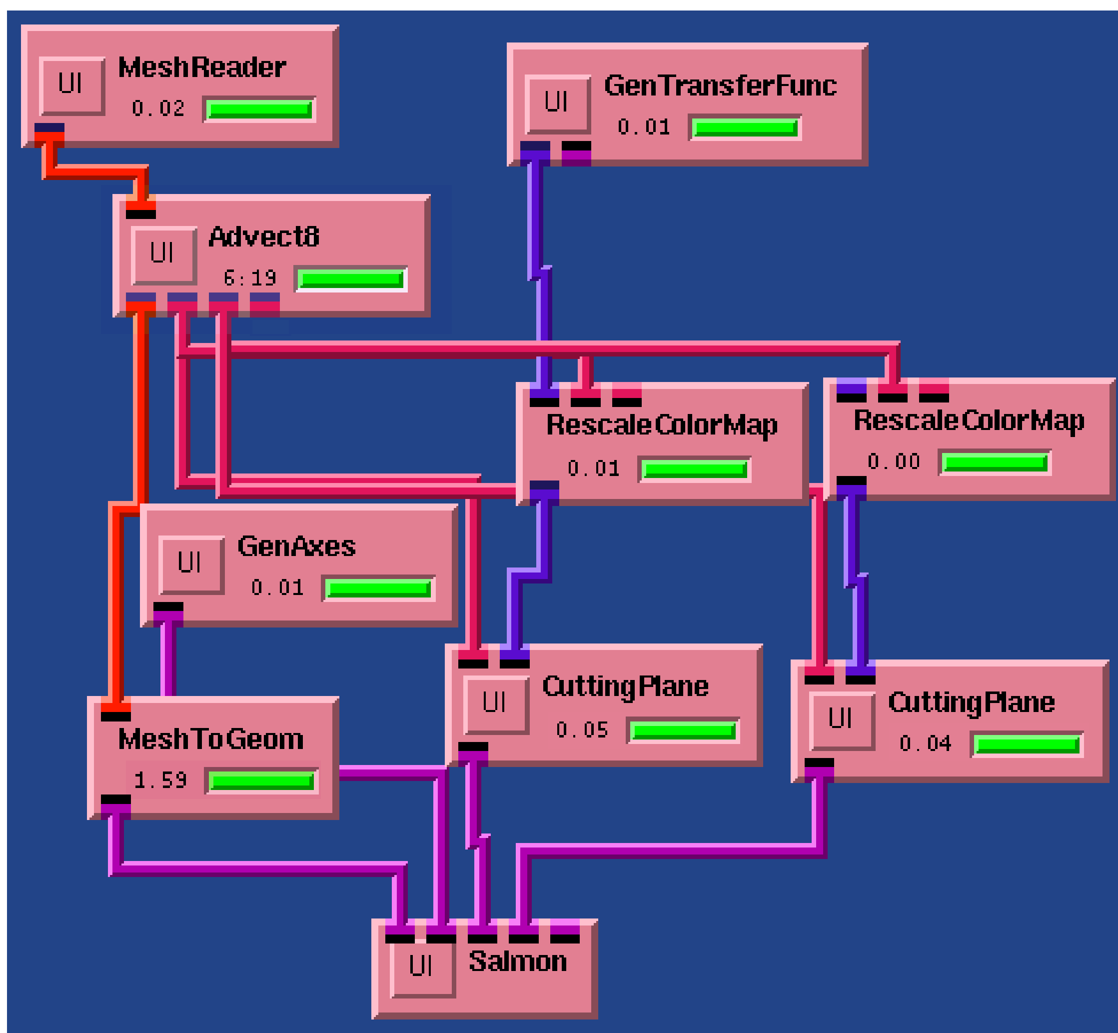

Figure 8.9. A SCIRun network for an example atmospheric diffusion simulation.

we demonstrated how a legacy Fortran application can be integrated into SCIRun's computational steering environment. Finally, we exhibited a time-varying problems using adaptive unstructured meshes, applied to a model of pollution dispersion in the atmosphere. These problems represent only a few the applications which are possible with SCIRun. In each of these applications, SCIRun is used to provide interactivity that was previously not available in these long-running applications.

In the torso defibrillator example, a user is free to try various numerical algorithms, various mesh discretizations, and most importantly, various defibrillator electrode placements. The scientist is able to leverage the computational components that have been implemented in SCIRun modules. If a scientist were to implement a different simulation, a large number of these modules would not need to be rewritten. In addition, the scientist can actually interact with the simulation. In addition, the integrated environment allows several important optimizations to be made. Instead of regenerating the mesh for each new electrode configuration, the system simply updates the mesh in the regions were the electrode is placed. This can reduce the mesh generation process from multiple hours to a handful of seconds. In addition, some optimization is obtained through using the previous solution as an initial guess to the new solution. If small modifications are being made to the design, this can reduce the number of iterations that are required for convergence. This also displays a visually smooth transition from one solution to another.

For the Monte Carlo global illumination model, interaction is more limited. However, it demonstrated how a long-running application can use the fine-grained dataflow aspects of SCIRun to provide incremental updates to a long-running program. While the path tracer module is computing the image, the user can manipulate parameters in downstream modules. In addition, the user can modify adaptation criteria and sample densities while the path tracer module is still running. This is not possible in the typical stand-alone application. A program can be custom built that would perform the same tasks. However, this program would be customized for this particular application. The power of SCIRun is utilized when components created for one application can be reused in another application.

The CFDLIB example demonstrated how these ideas can be implemented for a legacy

program. Small modifications were made to the source code of the application that allowed SCIRun to extract data at appropriate points in the computational process. These modifications made subroutine calls to build SCIRun data structures and pass them along down through dataflow network. Whereas this is possible in existing dataflow systems, SCIRun adds a new degree of flexibility. Typically, the CFDLIB internal data structures would be translated to the required internal format of the visualization package. Since CFDLIB uses cell-centered data variables, the data values would have to be interpolated back to the cell corners for most visualization packages. This translation is typically expensive, and adds to the perceived cost of visualizing the data during runtime. In some cases, translating to the required data formats of visualization packages can also introduce error or reduce the accuracy of the represented data.

Finally, the atmospheric diffusion example demonstrated another legacy program which also performed a time-varying simulation. An adaptive unstructured mesh used the unstructured grid aspects of the SCIRun modules. In addition, it demonstrated that SCIRun can be more than a visualization tool when used with existing applications. In this example, A SCIRun module (the initial mesh generator) was used as an input to the TETRAD simulation.

8.6 Comparison with Existing Systems

In addition to the applications, there are important differences in the flexibility and performance of SCIRun and comparable dataflow systems. These performance differences become especially important when one tries to perform large-scale simulations in an interactive setting.

8.6.1 Performance

SCIRun seeks to build on the conceptual successes of currently dataflow systems, yet expand the scope and performance over available systems. Since this system is designed to support large-scale problems, efficient means of storing and transferring data and powerful, robust algorithms have been the focus of development throughout.

The modularity of visual dataflow environments and with the interactivity they support makes this paradigm attractive for problem solving. However, the visual program-

ming systems mentioned in Chapter 2 (Iris Explorer, AVS, and DX) have proven inadequate for solving many large-scale scientific and engineering field problems [14]. These systems were designed specifically for visualization and lack integrated tools to support modeling and simulation. These systems also lack system utilities for monitoring and controlling system resource utilization, such as memory and processor usage. Moreover, they tend to duplicate the geometric model and simulation data as they pass between modules—a reasonable property as long as the datasets are small, but a significant bottleneck in large-scale computation. For example, replication is not reasonable when the data contains multimillion element meshes or a stiffness matrix with millions of entries. Finally, these systems, although supporting many general purpose visualization methods, lack tools for efficiently visualizing large data sets.

Isosurface extraction is an excellent example of a task that can become very computational intensive for large datasets. Table 8.1 shows memory requirements and execution times for extracting an isosurface of electric potential from a 420,000-element tetrahedral volume mesh. These data come from a simulation of the electric field induced by defibrillator patch electrodes placed within a geometric model of a thorax near the epicardial surface, as described in Section 8.1. DX, Iris Explorer, AVS, and SCIRun were all able to construct the same isosurface. However, as the timings in the table demonstrate, this is not an interactive process for the first three. Having been designed since inception

Table 8.1. Benchmarks of sample PSEs

Dataflow System	Memory Usage	CPU time
AVS 5.x	89 MB	3 sec
NAG Iris Explorer	72 MB	20 sec
IBM Visualization Data Explorer	63 MB	2 sec
SCIRun	63 MB	0.6 sec

This table shows memory consumption and execution times for an isosurface extraction operation using SCIRun, AVS 5.x, Iris Explorer and IBM Data Explorer. These benchmarks were performed on a 175Mhz Silicone Graphics Octane R10000.

to perform efficiently on large-scale problems, SCIRun is able to routinely outperform these commercial systems.

The most noticeable shortcoming of AVS 5.x over all the other systems is its large memory usage. This arises from the undesirable property that for each additional modules appended to the visualization sequence, a further copy of the dataset is made. To illustrate this effect, we added first a gradient module, which computes the spatial gradient of the scalar field, and then a streamline module and a "hedgehog" module to visualize the gradient field. The memory usage of SCIRun went up to a modest 81 MB, whereas the memory usage of AVS rose to over 153 MB. This trend is illustrated in Table 8.2. A network with a wide array of modeling, simulation and visualization models would further exacerbate this problem.

Iris Explorer uses shared memory to communicate between modules, and thus its memory usage is lower than that of AVS. However, since each module within Iris Explorer is essentially its own process—its own independent program, each allocate its own private data separate from the shared pool. This separation of memory areas presents considerable complexity for the developer of new modules because of the need to manage two separate memory pools. This is especially problematic when attempting to incorporate existing program source code, which typically assumes a single memory area, into an Iris Explorer module. For unknown reasons, the isosurface extraction in Iris Explorer is also considerably slower than the others.

IBM Data Explorer came closest to the performance of SCIRun. Its memory usage is virtually the same as SCIRun, because of the cache management algorithm it employs.

Table 8.2. PSE memory usage

Modules used	AVS 5.x	SCIRun
Isosurface	89 MB	63 MB
Isosurface + gradient + streamline	126 MB	80 MB
Isosurface + gradient + streamline + hedgehog	153 MB	81 MB

Memory usage of AVS 5.x and SCIRun as a function of the number of visualization steps in the network of modules.

Execution time is somewhat higher than that of SCIRun but, at least for this dataset, within an acceptable range for interactive use.

8.6.2 Flexibility

One of the goals in any modular programming environment is to create elements that can easily be used for many purposes. A high level of reuse ensures that an element is well tested, under a wide range of conditions, and thus becomes, over time, a robust and stable piece of software. One way to increase the degree of reusability of software is to make each modular element as flexible as possible, so that it can be used under a variety of conditions without modification to its core functionality.

SCIRun makes use of some features of object-oriented programming to achieve a high level of code flexibility and thus reusability. In an object-oriented data model, pieces of data are thought of as objects upon which computations are executed. A powerful property of objects is that they can be specialized, or *derived,* from a more general object into variants with slightly differing functionality. This occurs in a way that inherits all of the content and functionality of the parent object, hence removing the need to recreate this functionality each time a new object is derived. In SCIRun, a user can easily introduce a new, specialized type of object, without having to alter any other part of the system that uses the same general type. This level of flexibility ensures maximum reuse of essential elements and allows development efforts to focus on these critical components instead of having multiple versions to create and maintain.

Few of the commercial PSE programs provide this same level of flexibility and reusability in their data structures. For example, AVS 5.x has two different data models, known as “fields” (block structured grids) and “ucd” (unstructured grids). Each data type requires its own module, resulting in a doubling of the number of modules that much be developed and maintained. Furthermore, the functionality of the two different versions of each module typically differs dramatically. For example, the ucd streamline module provides support for stream ribbons, whereas the normal streamline module does not. Iris Explorer suffers from this same weakness. Functionality of the modules based on, for example, unstructured grids tends to lack far behind that of structured grids.

With SCIRun, the same streamline module can be used on any type of vector field.

This is true even for field representations which were not implemented at the time which the Streamline module was compiled.

IBM Data Explorer avoids this problem by providing a unifying data model. Although this does improve flexibility, it goes perhaps too far by forcing *all* sources of data of any sort to have the same structure. Since computations of a field has a much different structure than visualization of the geometric model, the unified structure almost always requires additional steps to extract and convert data into the appropriate form for internal use, thus adding more overhead and programming complexity to each module. It also creates overhead of converting to the common form when integrating codes into the environment that do not exactly match this common form. This overhead is acceptable when the system is used in a visualization mode only, since the data are typically converted once and stored into a file. However, in a computational steering system, the data would be converted at each time step, or after each change.

All three of the commercial systems use a general unstructured grid data model. We suspect that this also contributes considerably to the performance differences exhibited in Table 8.1. SCIRun attempts to strike a balance between unified and specialized data modules by supporting a few basic data types, from which others can be derived. SCIRun is able to work directly on the appropriate data in an appropriate form, reducing the need for conversion or extra storage between elements.

8.7 Chapter Summary

We described four applications that demonstrate the successful application of the concepts proposed by this thesis. A computational medicine application (torso defibrillator modeling) demonstrates a large-scale finite element design simulation, implemented entirely with the SCIRun components. A computer graphics application (Monte Carlo global illumination) demonstrates the utility of fine-grained dataflow in a long-running simulation. A computational fluid dynamics simulation demonstrated the combination of existing code with the SCIRun computational steering tools. The final application, TETRAD, demonstrates a time-varying simulation with adaptive finite-element meshes. Finally, we made a comparison of performance between SCIRun and other cur-

rent dataflow-oriented visualization tools. Here we demonstrate that SCIRun attains substantial memory and speed improvements over existing packages. These improvements become even more important as we use the environment for modeling and computation components in addition to the visualization tools provides by these systems.

CHAPTER 9

CONCLUSIONS

This dissertation demonstrates that computational steering can be a useful tool in computational science, engineering, and medicine applications and that this utility is obtained by providing a flexible and efficient infrastructure where disparate computational tools can be used in a single focused environment. Furthermore, we demonstrate that by using such an environment, a scientist or engineer can rapidly investigate the solution space for iterative computational design problems.

The SCIRun system provides a problem solving environment for building scientific models, simulations, and scientific visualizations across many different application domains. It allows a scientist or engineer to interactively steer a computation, changing many different parameters, recomputing, and then revisualizing the results all within the same programming environment. SCIRun supplies application-specific modules, as well as generalized scientific datatypes and modules. Modules can be created from the existing code resources of potential users. SCIRun has been utilized for research within the domains of bioelectric field studies in cardiology and neuroscience, as well as computational geophysics and in other computational field problems.

The term *computational steering* refers to a user's ability to interact with an application's execution—to change the parameters of an algorithm in midexecution, without ever having to stop or restart computation. Such interactive capabilities contrast with the batch processing model, in which applications can be controlled by means of a list of start-up parameters, but once started, always either run to completion or are aborted in midstream. Coupled with a visual programming environment, computational steering is a powerful problem solving tool. Each module of a PSE contains a set of interface controls that the user can adjust at any time during execution. Each time a parameter

changes, its value propagates to any other modules that depend on either the parameter value or the output of any other connected module. By this propagation mechanism, the user is freed from the need to explicitly deal with implications of the parameter change but is still able to vary any parameter at any time during execution. Making changes to the parameters does not stop the computations in progress but guides, or steers, them through their execution. Through this combination of manual and automatic processing, a steerable PSE offers a simple, efficient means of operation while still maintaining a great deal of control and flexibility.

9.1 Contributions

The focus of this dissertation a module-based, dataflow-style visual programming environment. In addition to extensibility typically associated with this class of tools, it utilizes extensibility features of object-oriented programming. Modifications to the traditional dataflow implementation allow large-scale applications to be created and executed efficiently. In this environment, applications can be controlled through direct lightweight parameter changes, program cancellation, and feedback loops in the dataflow graph. A set of data modeling classes provides the basis whereby computational field problems can be constructed. These data classes and the computational modules are all implemented using a convenient set of support libraries that help a module writer to create simple yet robust modules.

The computational steering system presented here combines several different computer science concepts to achieve this flexibility and efficiency. A dataflow [27] visual programming model is combined with concepts from object-oriented programming to achieve higher modularity by insulating data representations from dataflow modules. This combination also provides a mechanism for implementing demand driven dataflow (lazy evaluation) [120] in a straightforward manner. Threads [111] are used to provide task parallelism expressed naturally in the dataflow graph, as well as data parallelism coded explicitly in different components. Finally, we extend the concept of dataflow's internal communication by allowing more flexible communication ports. This facilitates a more expressive set of communication methods for implementing relationships that do

not fit naturally in the dataflow model. These ports also provides a mechanism for using fine-grained dataflow mixed with coarse-grained dataflow in a flexible manner.

SCIRun offers a visual programming environment for scientific computing. It differs from traditional dataflow-oriented visual programming systems in that it focuses on efficiency for large-scale computational problems. SCIRun extends the typical dataflow system by combining concepts from object-oriented programming and by generalizing dataflow communication ports to allow for component relationships which are difficult to express using the dataflow metaphor. One essential contribution is that this efficiency can be achieved while maintaining the simple composability typically associated with dataflow toolkits.

The choices made in the implementation of SCIRun address the following issues:

- **Interactivity.** Ideally, all computation would be instantaneous. In reality, however, many large-scale scientific applications take minutes to hours to days to complete. Where appropriate, SCIRun uses intermediate results to allow the user to examine a computation in progress. SCIRun also operates efficiently so that in those cases where a degree of interactivity is attainable the computation will not be impeded by the system.

- **Integration.** The modeling, simulation, and visualization aspects of the problem are integrated within SCIRun. SCIRun does not make special distinction between these different types of components. In designing the representations of the SCIRun data models, we have addressed the requirements imposed by these different components. Finally, the SCIRun dataflow programming model uses these components together to provide solutions to a scientific or engineering problem.

- **Extensiblity.** SCIRun extends the traditional dataflow model of extensibility by combining it with object-oriented based methods for extensibility. Through the combination of these features, SCIRun can be used to add new visualization tools to existing simulations or to add new simulations to existing visualization tools. SCIRun does not attempt to be a monolithic solution for a handful of problems nor a completely general solution for all problems. It attempts to strike an efficient

balance between those two extremes.

SCIRun was one of the first systems to suggest that computational steering was most appropriate for iterative design problems, namely, for determining answers to "what-if?" questions in a variety of contexts [59]. With a cohesive integrated environment, SCIRun components can exploit spatial and temporal coherence (explained in Chapter 3) to reduce the computational cost for iterative analysis. The first computation is performed at a fixed cost, but subsequent iterations can be faster.

9.2 Pros and cons of the SCIRun approach

Four applications were demonstrated: Torso defibrillator modeling, Monte Carlo global illumination, a CFD application using CFDLIB, and a simulation of atmospheric diffusion. These applications were selected to explore the gamut of SCIRun's possibilities. However, many more applications can benefit from the SCIRun environment.

Steering a large scientific application involves much more than attaching a graphical user interface to a few parameters. Several of the systems mentioned in Chapter 2 have suggested excellent methods for extracting information from running programs, for injecting updates back into the program, and for managing these changes. We argue that these techniques are most effective when used in a highly integrated environment, where data can be shared among the various computing and visualization tasks.

Although computational steering was proposed many years ago, it has not been rapidly adopted among most scientists and engineerings. The primary reason that we have heard voiced is *Computational steering uses too much memory or CPU time and my code runs slower*. This has been addressed by the efficiency considerations that have been examined in SCIRun. In some cases, the resulting program can execute faster than the original stand-alone program, when it can take advantage of the coherence between user interactions.

In addition, scientists often say *I just want to focus on the science* or *I just want to focus on the visualization*. The modular SCIRun environment allows the scientist to use visualization modules without knowing much about their implementation. Similarly, the visualization programmer can write modules that visualize a wide variety of data without

knowing the specific details of a broad range of scientific data formats. Scientists can use modules outside of the domain of their expertise yet integrate those pieces with their most recent research programs.

The tightly integrated modular environment provided by SCIRun allows computational steering to be applied to a broad range of advanced scientific computations.

The concepts embodied in SCIRun are an attempt to strike a reasonable balance in a wide range of possibilities. Certainly other options can be considered.

Many of the systems for steering described in Chapter 2 have suggested good mechanisms for extracting data from running code, especially in the context of programs that already exist [15, 36, 49, 66, 124, 125, 126]. Many of these tools or concepts could be applied to computational steering in the SCIRun environment. Sensors and actuators [126] would provide a more cohesive set of mechanisms for extracting data from running code and for triggering changes in program variables. They would be a useful addition to an environment such as SCIRun.

In particular, SCIRun requires that modules be explicitly written to support steering. The dataflow structure automatically provides a limited degree of steering (changing inputs and outputs), but the other mechanisms presented (such as changing user interface parameters) require explicit support in the module. In a design environment, this allows the module writer to limit the changes to those that are valid. However, during the development phases of a module it may be desirable to allow access to any part of the module. In SCIRun, we have delegated this role to the command line and graphical debuggers available on the operating system.

There are relationships that are difficult to cast in a dataflow network. Time-dependent problems, although presented in Chapter 8, still present some difficulty. Feedback loops, described in Chapter 4, are often difficult to control. Finally, dataflow is good at representing "producer-consumer" relationships. We extended the ports in Chapter 4 to also support "client-server" relationship. However, in object-oriented programming, the "uses" relationship [68] is also common. In the future, we plan to investigate the possibility of further generalizing the dataflow port concepts to support such relationships.

Many of these issues will be addressed (and possibly new drawbacks introduced) as

SCIRun continues to evolve.

As we have seen, steering a large scientific application involves much more than connecting a graphical user interface to a few parameters. SCIRun employs several techniques exist for extracting information from running programs, for injecting updates back into the program, and for managing these changes. Each of these techniques is useful in different contexts for different types of parameters. The highly integrated environment provided by SCIRun allows these techniques to be used in chorus.

CHAPTER 10

ONGOING AND FUTURE WORK

In a flexible, extensible environment such as SCIRun, there are numerous possibilities for expansion. Here we describe several applications that are currently using SCIRun. Modules and data structures are being created to support these applications specifically, and the current infrastructure will be improved to meet these demands.

One of the largest infrastructure changes will be support for execution in a networked/distributed environment [81, 82]. For simplicity, we focused on shared memory multiprocessors for the initial implementation of SCIRun. Along with this will come “detachable user interfaces.” Currently SCIRun applications must be executed within the SCIRun user interface, but for a long-running simulation, it would be beneficial to start the program and then come back periodically to check on the progress of the program. The user can steer the simulation and then return at a later time to see the effects of the changes. This modification would likely be performed in conjunction with a modified user interface that reduces the number of popup windows scattered about the user’s screen.

Many of these projects are being tackled by current Ph.D. and M.S. candidates.

10.1 Common Component Architecture

This is a current and ongoing collaboration between the Center for Scientific Computing and Imaging (SCI) at the University of Utah and the Department of Energy National Laboratories (and other university research groups). With representatives from these facilities, the Common Component Architecture (CCA) Working Group was formed “to develop a specification for a component architecture for high-performance computing.” This goal has long been a central theme of the SCIRun problem solving environment. The proposed project will form a symbiotic relationship between the DOE labs, Utah,

and other university participants, in which SCIRun will provide applications, tools, and experience with high-performance component architectures to the CCA community, and the DOE labs and other university groups part of the CCA effort will help provide SCIRun with a more flexible and widely accepted component model.

The CCA Working Group was established in the beginning of 1998 by researchers who develop component architectures for their respective institutions. The mission statement from the CCA working group is [22]:

> Our object is to develop a specification for a component architecture for high-performance computing. The CCA's intent is to provide a pan-DOE lab framework that would provide a plug and play interface for the DOE's many high performance numerical components and tools. The CCA is not restricted to DOE, however and universities and commercial labs are welcome to participate. Further, it is our intent that these components can be mixed and matched to create high-performance applications on the fastest and highest-capacity machines on the planet. Examples of these components are equation solvers, explicit stencil solvers, and attendent load-balancers; everything that is needed to compose a high-performance simulation and make it go.

Since that goal was set, the CCA working group has proceeded to define the component architecture and is in the process of solidifying a specification. This specification will enable us to cast the component model used in SCIRun into a more standard form. When complete, components from SCIRun can be used in other frameworks, and components from these frameworks can be used within the SCIRun framework. SCIRun will continue to grow and evolve based on this collaboration.

10.2 C-SAFE

The University of Utah has created an alliance with the DOE Accelerated Strategic Computing Initiative (ASCI) to create the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [6]. It focuses specifically on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions, especially within the context of handling and storage of highly flammable materials. The objective of the C-SAFE is to provide a system comprising a problem solving environment in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers, optimization, computational steering, visualization, and experimental

data verification. The availability of simulations using this system will help to better evaluate the risks and safety issues associated with fires and explosions. Our team will integrate and deliver a system that will be validated and documented for practical application to accidents involving both hydrocarbon and energetic materials. This system will be based on SCIRun.

Although the ultimate C-SAFE goal is to simulate fires involving a diverse range of accident scenarios including multiple high-energy devices, complex building/surroundings geometries, and many fuel sources, the initial efforts will focus on the computation of three scenarios:

- rapid heating of a container with conventional explosives in a pool fire (e.g., a conventional bomb involved in an intense jet-fuel fire after an airplane crash),
- impact and ignition of a container with subsequent explosion and firespread (e.g., shelling of a mine storage building by terrorists),
- heterogeneous fire containing a high energy device (e.g., ignition of a containment building in a missile storage area).

These large-scale problems require consideration of fundamental gas and condensed phase chemistry, structural mechanics, turbulent reacting flows, convective and radiative heat transfer, and mass transfer, in a time-accurate, full-physics simulation of accidental fires. This simulation will be expansive enough to include the physical and chemical changes in containment vessels and structures, the mechanical stress and rupture of the container, and the chemistry and physics of organic, metallic, and energetic material inside the vessel. The simulation will include deflagration-to-detonation transitions (DDT) of any energetic material in the fire, but the simulation will end when/if detonation occurs. C-SAFE will provide coupling of the micro-scale and meso-scale contributions to the macroscopic application in order to provide full-physics across the breadth of supporting mechanistic disciplines and to achieve efficient utilization of ASCI program supercomputers.

10.3 Parallel CSPRINT/Tetrad

The Tetrad application described in Chapter 3 and Chapter 8 has been parallelized using MPI. This presents a challenge to SCIRun, since SCIRun has focused on shared-memory machines. Efforts are underway to extend SCIRun to a distributed memory machine. The parallel version of Tetrad is a driving application for this effort.

10.4 Neuroscience Inverse Problem

Neuroscience inverse problem: Excitation currents in the brain produce an electrical field that can be detected as small voltages on the scalp. By measuring changes in the patterns of the scalp's electrical activity, physicians can detect some forms of neurological disorders. Electroencephalograms (EEGs) measure these voltages; however, they provide physicians with only a snapshot of brain activity. These glimpses help doctors spot disorders but are sometimes insufficient for diagnosing them. For the latter, doctors turn to other techniques, in rare cases to investigative surgery.

Such is the case with some forms of epilepsy. To determine whether a patient who is not responding to medication has an operable form of the disorder, known as temporal lobe epilepsy, neurosurgeons use an inverse procedure to identify whether the abnormal electrical activity is highly localized (thus operable) or diffused over the entire brain.

The epileptic foci are represented as a set of idealized dipole sources situated in the temporal lobe. Using a model of the human skull and brain, the direct EEG problem is posed by solving generalized Poisson equations for the voltage and current distribution within the brain and upon the surface of the scalp.

10.5 Inverse-EEG Pipeline

The inverse-EEG problem [129] can be described as the mathematical mapping of EEG scalp recordings back onto the cortical surface or within the cortex to approximate fundamental current sources. This inverse problem lies at the foundation of surgical planning and prognosis for neurological conditions ranging from epilepsy to schizophrenia [34] and to brain tumors. The goal of cortical mapping is to integrate patient anatomic information and measured voltage potential recordings from the surface of the patient's

scalp in order to noninvasively determine the electrical activity on and within the patient's cortical surface [74].

10.6 Optical Tomography

In collaboration with Martin Schwieger and Simon Arridge at the University College London (UCL), we have utlized UCL imaging package for absorption and scatter reconstruction, from time-resolved data (TOAST) [118], and embedded it in the SCIRun interactive simulation and visualization package developed at the University of Utah [108]. Reconstruction of a segmented 3D head model is used as an example for demonstrating the capabilities of the combined TOAST/SCIRun approach. The primary reason for integrating TOAST into SCIRun is to enable the user to interactively control the modeling, simulation, and visualization parameters — even while the computation is in progress. This control allows the user to vary boundary conditions or model geometries or various computational parameters during the simulation. SCIRun provides full visualization capabilities together with fast and efficient matrix solvers.

We have performed a preliminary study on the use of advanced 3D Finite Element modeling, reconstruction, and visualization TOAST-SCIRun software for Optical Tomography. We should emphasis that several aspects of the case modelled here are too simplistic, but the ease of integration of existing software and the tractable reconstruction times obtained suggest that such an approach is promising. We are currently addressing better meshing strategies, as well as improved reconstruction methods. Furthermore, more interactive links will be integrated into the TOAST-SCIRun system to take better advantage of the SCIRun steering capabilities.

10.7 Visualization and Manipulation of Large-scale Datasets

In addition to the medical simulations described here, SCIRun is also being used to perform a wide variety of medical visualizations. Since SCIRun is designed for large-scale problems, it is being successfully applied to visualize large medical imaging datasets, even when there are not significant components of simulation and modeling. These datasets are becoming larger at a rapid pace and are quickly outstripping the

capabilities of currently available systems. Specifically, SCIRun is being used as a basis for some of the research performed by the Advanced Visualization Technology Center (http://www.avtc.org). This is a collaboration between the University of Utah, Los Alamos National Laboratory, and Argonne National Laboratory for developing methods to interactively visualize and steer multiterabyte datasets.

10.8 Local and Global Visualization Using Haptic Devices

Currently, most graphical techniques emphasize either a global or local perspective when visualizing vector or scalar field data, yet ideally one wishes simultaneous access to both perspectives. The global perspective is required for navigation and development of an overall gestalt, whereas a local perspective is required for detailed information extraction. Our approach is to augment a global visual display with local display methods that combine graphics and haptic displays.

- Using global visualization techniques such as line integral convolution (LIC) [21], volume rendering [62, 67, 80, 117] and critical point extraction [85], we enable simultaneous local and global field visualization by allowing the user to introduce flow advection icons such as streamlines, stream ribbons, streamtubes, and/or colored dyes into the global representation of the vector or scalar field.

- Using a haptic interface, such as the Sarcos Dextrous Arm Master [46] or Phantom [103], we will provide haptic representations of scalar and vector field intensities in local regions while visualizing the global field [87]. We will also investigate haptic methods to facilitate visualization of heterogeneous data, such as simultaneous scalar and vector fields.

In order to apply local and/or global visualization methods to large-scale 3D data sets in an interactive manner, we will need to develop new visualization algorithms. In addition, the human expert will need to have access to computational steering aids in order to close the loop of simulation and visualization. The haptic and visualization algorithms will be integrated into SCIRun to allow for rapid feedback when visualization parameters

are changed, such as the location of a 3D streamline advection widget or the method of haptic interaction such as mapping to joints or the control law for force display.

10.9 Chapter Summary

In this chapter we described several large-scale scientific, engineering, and medical applications that are being developed within the SCIRun problem solving environment. The flexible, extensible, integrated programming environment provided by SCIRun makes it a natural tool for implementing and interacting with such large-scale modeling, simulation, and visualization applications.

APPENDIX

SOFTWARE DESIGN

This appendix will describe many of the design choices that were made as we wrote SCIRun. In particular, we will describe design choices which were changed in the middle of the project, and design choices that we would reconsider if we were starting the project again from scratch.

A.1 The C++ Programming Language

In 1994, when we started the development of SCIRun, the C++ programming language was still very immature. In particular, many features of C++ had not been implemented in the available compilers.

In particular, the choice of C++ was a controversial choice for implementing scientific programs. Many have experienced difficulty with the performance of C++ when using it in this manner. In examining this issue, we found the the lack of performance stemmed from two issues:

1. Immature optimization in C++ compilers: We addressed this issue by providing C code kernels for those few portions of the code for which performance was most critical. In SCIRun, these were the code kernels mentioned in Chapter 6. Current compilers have addressed this issue to a large degree but have not yet reached perfection, so the need for the hand-tuned C kernels still exists.

2. Excessive use of abstraction: Many novices haphazardly use all of the nice abstractions which C++ provides. In many cases, they produce very readable programs which run very slowly. In SCIRun, we have attempted to recognize the costs of these abstractions and to use them appropriately.

The C++ programming language has been a good choice. However, newer features, such as exceptions, namespaces, and member templates, were not completely specified or not widely available at the time of SCIRun's inception. SCIRun will likely evolve to leverage many of these newer programming features.

A.2 Library: Multitask

When SCIRun started, no object-oriented thread libraries were available. Today, the Java language provides a nice interface to threads. Other libraries in C++ are also under development. In a future version of SCIRun, we will adopt a Thread library which is modeled after the Java thread library. It uses the term "Thread" instead of "Task," since that has become the term of choice.

A.3 Library: Classlib

The Standard Template Library (STL) has recently emerged as a powerful way of expressing standard container template objects. SCIRun may evolve to utilize the STL more completely and may adopt many of its methodologies (i.e., iterators). However, the STL has a very steep learning curve. It is also unclear how efficient the STL will be in a high performance computing environment.

Finally, a new implementation of SCIRun would make more extensive use of exceptions and namespaces which are now available in most C++ compilers.

A.4 Library: TCL

The very first implementation of SCIRun used a Motif-based user interface. It leveraged MiNT [24], an object-oriented wrapper to the Motif libraries. MiNT drastically reduced the complexity of implementing a Motif user interface, but the process was still very complex. In particular, callbacks were extremely cumbersome. One was required to implement a wealth of tiny callback functions in order to implement even the simplest user interfaces. Since Motif is not multithreaded, the Motif code had to be carefully managed with locks.

Several months after we started SCIRun, we adopted Tcl/Tk as the user interface. It was a drastic improvement over Motif. The primary advantage is the reduction in lines of

code required to implement a user interface. A secondary advantage was the availability of a scripting language, which we adopted for saving SCIRun networks, for evaluating expressions, among other uses. We were also able to rapidly prototype user interfaces — drastically reducing the development cycle time. The layout mechanisms available with the "pack" mechanism are much simpler than those found in the Motif form. Finally, Tcl/Tk seems to have a smaller memory footprint than Motif.

However, Tcl/Tk did not solve all problems with the user interfaces. Tcl/Tk is not thread-safe, so meticulous locking is still necessary. Tcl/Tk is currently very slow, which limits the rate at which the program can efficiently update the user interface. Since the user interface is used as a visualization tool to inform the user of the internal state of SCIRun, we would like to have a more flexible, more efficient interface that can perform these updates more frequently.

The use of Tcl/Tk also complicates the distribution of a SCIRun "binary," since the Tcl script files are required in order to execute SCIRun. However, this is not a major limitation since a single file could be created that contains all of the Tcl code. It could also be encrypted or otherwise encoded in order to protect the contents of the Tcl code.

A.5 Library: Dataflow

The "extended ports" described in Chapter 4 are not perfectly general. A newer system is under consideration where the system can replace the implementation of a port with different mechanisms (i.e., one for local communication, one for remote, etc.). The ports should also be able to express a "uses" relationship between modules, in addition to the client-server and producer-consumer relationships currently utilized.

Ideally, the visual programming interface would reflect these relationships in the dataflow network.

A.6 Library: Datatypes

Many of the Datatypes mentioned in Chapter 5 have outgrown their initial design. In particular, the Mesh class is likely to be reconsidered. A newer implementation will separate the nodes from the mesh, using a "NodeSet" container to hold the nodes. This will provide a mechanism to share nodes between multiple meshes and even surfaces.

In addition to this change, we will implement element iterators which will access the elements of the mesh for algorithms such as isosurfacing which currently subvert the type system. We will need to experiment with methods to make this efficient.

The Field classes are also likely candidates for redesign. They currently do not make use of templates which become cumbersome when trying to implement fields of different numerical representation (i.e., int vs. short vs. double, etc.), and the fields are not extensible to other types of fields (i.e., colors, tensors, etc.).

Finally, it will probably be a good idea to provide explicit 2D analogs to many of the 3D types provided in SCIRun. Currently, we embed 2D data in 3D fields, but it wastes at least a factor of two in data storage and it does not provide for the most efficient access (i.e., tri-linear interpolation instead of bilinear interpolation). As SCIRun becomes used for other 2D simulations, it makes sense to cure these ills. In addition, we will need to implement the 2D analogs to many of the modules, most notably a 2D graphical viewer.

A.7 Library: Geom and Module: Salmon

When SCIRun was initially implemented, two libraries were available commercially for developing 3D graphics applications. Iris Inventor [130] is a scenegraph based system for representing geometry. However, it is not thread-safe and it was not very efficient. More recent implementations have addressed the performance issues, but the issue of thread-safety made the library incompatible with the multithreaded SCIRun environment. The second system is Iris Performer [101, 112]. It was originally designed for multithreaded visual simulation systems, so does not suffer from the same limitation as Inventor. It was also designed for high performance. However, it is consumes a considerable percentage of processor resources. Early versions required that three processors be dedicated to the Performer rendering pipeline. This is not acceptable on a desktop system. Current versions now work on desktop systems.

SGI recognizes that neither of these products are sufficient for many systems, including SCIRun. SGI has commissioned several efforts to address this problem, including Cosmo 3D, OpenGL++, and the latest - Fahrenheit. When Fahrenheit materializes, it will be considered as a suitable replacement for the Geom library which currently suffers

from a lack of a coherent design. At the same time, the Salmon module will need to be reimplemented.

A.8 Compatibility with Commercial Systems

We chose to write SCIRun from scratch instead of trying to address the problems with currently available commercial systems. This allowed us greater freedom in choosing a solution that may not be compatible with existing commercial code. It was also important to have access to the source code of the implementation; commercial systems simply do not have APIs that support the type of changes that we have made with SCIRun.

However, there are many modules written for AVS, Iris Explorer, Data Explorer and the Visualization Toolkit which may be very useful in the SCIRun environment. If the project were started over, we would closely examine the possibility of providing support to utilize these existing modules. It is not clear if the work saved by writing the compatibility interfaces would be greater than the work saved by reusing the modules. However, the bigger issues surround the legal ramifications of such a system.

The Visualization Toolkit (vtk) would not suffer from such ramifications, since it is a public domain system. It has also been a frequently requested addition to SCIRun. Although it may not be as efficient as SCIRun in some cases, the combination would allow the adoption of techniques that were developed for vtk.

A.9 Appendix Summary

We have described some of the design decisions made while implementing SCIRun. Much of the computing world has changed since many of these design decisions were made, so some of the design many need to evolve. Many of the decisions made in the future will be driven by those applications described in Chapter 10.

REFERENCES

[1] http://www.mathworks.com/.

[2] http://www.mathematica.com/.

[3] http://www.maplesoft.com/.

[4] http://gnarly.lanl.gov/Cfdlib/Cfdlib.html.

[5] http://www.rational.com/.

[6] Center for the Simulation of Accidental Fires and Explosions at University of Utah: Executive Summary: http://www.csafe.utah.edu/execsum.html.

[7] *setjmp(3C) Unix manual page*.

[8] *VolPack User's Guide*. http://graphics.stanford.edu/software/volpack/.

[9] Publications of Stanislaw M. Ulam, Stanislaw Ulam 1909-1984. *Los Alamos Science* (November 1987), 313–317.

[10] AHMAD, I., AND BERZINS, M. An algorithm for ODEs from atmospheric dispersion problems. *Appl. Num. Math* (1997), 137–149.

[11] ALLEN, D., HAYNOR, D., AND BARDY, G. Visualization of 3-D finite element model analysis of defibrillation. In *Proceedings of the IEEE Engineering in Medicine and Biology Society 13th Annual International Conference* (1991), IEEE, pp. 774–775.

[12] ANDERSON, E., AND DONGARRA, J. Performance of LAPACK: A portable library of numerical linear algebra routines. *Proceedings of the IEEE 81*, 8 (1993).

[13] http://www.avs.com.

[14] BEAZLEY, D. *SWIG and the Creation of Scriptable Scientific Applications*. PhD thesis, University of Utah, Salt Lake City, Utah, 1998.

[15] BEAZLEY, D., AND LOMDAHL, P. Controlling the data glut in large-scale molecular-dynamics simulations. *Computers in Physics 11*, 3 (May/June 1997), 230–238.

[16] BOISVERT, R., AND RICE, J. From scientific software libraries to problem solving environments. *IEEE Computational Science and Engineering* (October 1996), 44–53.

[17] BROOKS, JR., F. P. "grasping reality through illusion - interactive graphics serving science". In *Proceedings of the Fifth Conference on Computers and Human Interaction* (May 1988), pp. 1–11.

[18] BURDEN, R., AND FAIRES, J. *Numerical Analysis*, fourth edition ed. PWS-KENT, 1989.

[19] BURNETT, M., HOSSLI, R., PULLIAM, T., VANVOORST, B., AND YANG, X. Toward visual programming languages for steering scientific computations. *IEEE Computational Science and Engineering 1*, 4 (1994), 44–62.

[20] BUTENHOF, D. *Programming with Posix Threads*. Addison-Wesley, 1997.

[21] CABRAL, B., AND LEEDOM, C. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH 93* (1993), ACM SIGGRAPH, pp. 263–270.

[22] Common component architecture forum. http://z.ca.sandia.gov/ cca-forum.

[23] CIARLET, P., AND LIONS, J. *Handbook of Numerical Analysis: Finite Element Methods*, vol. 1. North-Holland, Amsterdam, 1991.

[24] CLARK, M. MiNT: A C++ framework for writing distributed graphical applications using Open Software Foundation/Motif. Master's thesis, University of Utah, 1997.

[25] COHEN, M. F., AND GREENBERG, D. P. The Hemi-Cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings) 19*, 3 (July 1985), 31–40.

[26] CONNER, D., SNIBBE, S., HERNDON, K., ROBBINS, D., ZELEZNIK, R., AND VAN DAM, A. Three-dimensional widgets. In *Computer Graphics (Proceedings of the 1992 Symposium on Interactive 3D Graphics)* (1992), ACM, pp. 183–188.

[27] DAVIS, A., AND KELLER, R. Data flow program graphs. *Computer* (February 1982), 26–41.

[28] DEROSE, T. A coordinate-free approach to geometric programming. *Theory and Practice of Geometry Modeling* (1989), 291–305.

[29] DEROSE, T. A coordinate-free approach to geometry programming. Tech. Rep. 89-09-16, University of Washington, September 1989.

[30] DONGARRA, J., LUMSDAINE, A., POZO, R., AND REMINGTON, K. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference* (1994), pp. 214–218.

[31] http://www.almaden.ibm.com/dx/.

[32] ET AL, T. D. F. Special issue on visualization in scientific computing. *Computer Graphics 21*, 6 (Nov. 1987).

[33] ETTER, D. *Structured Fortran 77 for Engineers and Scientists*, second ed. The Benjamin/Cummings Publishing Company, Inc., 1987.

[34] FAUX, S., MCCARLEY, R., NESTOR, P., SHENTON, M., POLLAK, S., PENHUME, V., MONDROW, E., MARCY, B., PETERSON, A., HORVATH, T., AND DAVIS, K. P300 topographic assymetries are present in unmedicated schizophrenics. *Electroencephalography and clinical Neurophysiology 88* (1993), 32–41.

[35] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[36] GEIST, G., KOHL, J., AND PAPADOPOULOS, P. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. *SIAM* (Aug. 1996).

[37] GRAVE, M., LE LOUS, Y., AND HEWITT, Eds. *Visualization in Scientific Computing*. Springer-Verlag, 199.

[38] GREENBERG, D., TORRANCE, K., SHIRLEY, P., ARVO, J., FERWERDA, J., PATTANAIK, S., LAFORTUNE, E., WALTER, B., FOO, S.-C., AND TRUMBORE, B. A framework for realistic image synthesis. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 477–494.

[39] GU, W., EISENHAUER, G., KRAMER, E., SCHWAN, K., STASKO, J., AND VETTER, J. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of the 5th Symposium of the Frontiers of Massively Parallel Computing* (Feb. 1995), pp. 422–429.

[40] GU, W., VETTER, J., AND SCHWAN, K. An annotated bibliography of interactive program steering. Tech. rep., Georgia Institute of Technology, 1994.

[41] HALL, R. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, 1989.

[42] HAMMING, R. *Numerical Methods for Scientists and Engineers*. McGraw-Hill, 1962.

[43] HART, G., TOMLIN, A., SMITH, J., AND BERZINS, M. Multi-scale atmospheric dispersion modelling by the use of adaptive grid techniques. *Environmental Monitoring and Assessment* (1997).

[44] HERNDON, K., AND MEYER, T. 3D widgets for exploratory scientific visualization. In *Proceedings of UIST '94, ACM Siggraph* (November 1994), ACM, pp. 69–70.

[45] HIBBARD, W., AND SANTEK, D. Interactivity is the key. In *Proceedings of the Chapel Hill Workshop on Volume Visualization* (May 1989), pp. 39–43.

[46] HOLLERBACH, J., AND JACOBSEN, S. Haptic interfaces for teleoperation and virtual environments. In *First Workshop on Simulation and Interaction in Virtual*

Environments (Iowa City, IA, July 13-15 1995).

[47] INC., S. M. XDR: External data representation standard. Tech. Rep. RFC 1014, Sun Microsystems, Inc., June 1987.

[48] http://www.nag.co.uk/Welcome_IEC.html.

[49] JABLONOWSKI, D., BRUNER, J., BLISS, B., AND HABER, R. VASE: The visualization and application steering environment. In *Proceedings of Supercomputing '93* (1993), IEEE Computer Society Press, pp. 560–569.

[50] JEAN, Y., KINDLER, T., RIBARSKY, W., GU, W., EISENHAUER, G., SCHWAN, K., AND ALYEA, F. An integrated approach for steering, visualization, and analysis of atmospheric simulations. In *IEEE Visualization '95* (1992), pp. 411–418.

[51] JOHNSON, C. Computational and numerical methods for bioelectric field problems. *Critical Reviews in BioMedical Engineering 25*, 1 (1997), 1–81.

[52] JOHNSON, C., BERZINS, M., ZHUKOV, L., AND COFFEY, R. Scirun: Application to atmospheric dispersion problems using unstructured meshes. In *Numerical Methods for Fluid Dynamics VI.* (1998), M. Baines, Ed.

[53] JOHNSON, C., AND MACLEOD, R. Computer models for calculating electric and potential fields in the human thorax. *Ann. Biomed. Eng. 19* (1991), 620. Proceedings of the 1991 Annual Fall Meeting of the Biomedical Engineering Society.

[54] JOHNSON, C., AND MACLEOD, R. Computer models for calculating transthoracic current flow. In *Proceedings of the IEEE Engineering in Medicine and Biology Society 13th Annual International Conference* (1991), IEEE Press, pp. 768–769.

[55] JOHNSON, C., AND MACLEOD, R. Nonuniform spatial mesh adaptation using a posteriori error estimates: applications to forward and inverse problems. *Applied Numerical Mathematics 14* (1994), 311–326.

[56] JOHNSON, C., MACLEOD, R., AND ERSHLER, P. A computer model for the study of electrical current flow in the human thorax. *Computers in Biology and Medicine 22*, 3 (1992), 305–323.

[57] JOHNSON, C., MACLEOD, R., AND MATHESON, M. Computer simulations reveal complexity of electrical activity in the human thorax. *Computers in Physics 6*, 3 (May/June 1992), 230–237.

[58] JOHNSON, C., MACLEOD, R., AND MATHESON, M. Computational medicine: Bioelectric field problems. *IEEE COMPUTER* (Oct., 1993), 59–67.

[59] JOHNSON, C., AND PARKER, S. A computational steering model applied to problems in medicine. In *Supercomputing '94* (1994), IEEE Press, pp. 540–549.

[60] JOHNSON, C., AND PARKER, S. Applications in computational medicine using SCIRun: A computational steering programming environment. In *Supercomputer '95* (1995), Springer-Verlag, pp. 2–19.

[61] KAJIYA, J. T. The rendering equation. *Computer Graphics 20*, 4 (August 1986), 143–150. ACM Siggraph '86 Conference Proceedings.

[62] KAUFMAN, A. *Volume Visualization*. IEEE CS Press, Los Alamitos, CA, 1990.

[63] KELLER, R. Semantics and applications of function graphs. Tech. Rep. UUCS-80-112, Computer Science Dept., University of Utah, Salt Lake City, Utah, 1980.

[64] KELLER, R., AND YEN, W.-C. A graphical approach to software development using function graphs. In *Digest of Papers Compcon Spring '81* (February 1981), pp. 156–161.

[65] KERLICK, G., AND KIRBY, E. Towards interactive steering, visualization, and animation of unsteady finite element simulations. In *Proc. Visualization '93* (1993), pp. 374–377.

[66] KOHL, J., PAPADOPOULOS, P., AND GEIST, G. CUMULVS: Collaborative infrastructure for developing distributed simulations. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing* (Mar. 1997), Minneapolis.

[67] LACROUTE, P., AND LEVOY, M. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24-29, 1994)* (1994), pp. 451–458.

[68] LAKOS, J. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[69] LANGTANGEN, H. Diffpack: Software for partial differential equations. In *Proceedings of the 2nd Annual Conference on Object Oriented Numerics (OON-SKI)* (1994).

[70] LERMAN, B., AND DEALE, O. Relation between transcardiac and transthoracic current during defibrillation in humans. *Circulation Research 67* (1990), 1420–1426.

[71] LEWIS, B., AND BERG, D. *Threads Primer: A Guide to Solaris Multithreaded Programming*. Prentice Hall, 1995.

[72] LIVNAT, Y., SHEN, H., AND JOHNSON, C. A near optimal isosurface extraction algorithm using the span space. *IEEE Transaction on Visualization and Computer Graphics 2*, 1 (March 1996), 201–227.

[73] LORENSEN, W., AND CLINE, H. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics 21*, 4 (1987), 163–169.

[74] LUTKENHONER, B., MENNINGHAUS, E., STEINSTRATER, O., WIENBRUCH,

C., GISSLER, M., AND ELBERT, T. Neuromagnetic source analysis using magnetic resonance images for the construction of source and volume conductor model. *Brain Topography 7*, 4 (1995), 291–299.

[75] LUTZ, M. *Programming Python*. O'Reilly and Associates, Sebastopol, CA, 1996.

[76] MACLEOD, R., JOHNSON, C., AND MATHESON, M. Visualization of cardiac bioelectricity — a case study. In *IEEE Visualization '92* (1992), pp. 411–418.

[77] MACLEOD, R., JOHNSON, C., AND MATHESON, M. Visualization tools for computational electrocardiography. In *Visualization in Biomedical Computing* (1992), pp. 433–444.

[78] MACLEOD, R., JOHNSON, C., AND MATHESON, M. Visualizing bioelectric fields. *IEEE Computer Graphics and Applications* (1993), 10–12.

[79] MARSHALL, R., KEMPF, J., DYER, D., AND YEN, C.-C. Visualization methods and simulation steering a 3D Turbulence model of Lake Erie. In *1990 Symp. on Interactive 3D Graphics* (1990), pp. 89–97.

[80] MAX, N., CRAWFIS, R., AND WILLIAMS, D. Visualizing wind velocities by advecting cloud textures. In *Proceedings of Visualization '92* (1992), IEEE Computer Society Press, Los Alamitos, CA, pp. 171–178.

[81] MILLER, M., HANSEN, C., AND JOHNSON, C. *Lecture Notes in Computer Science*. Springer-Verlag, 1998, ch. Simulation steering with SCIRun in a distributed environment.

[82] MILLER, M., HANSEN, C., PARKER, S., AND JOHNSON, C. Simulation steering with scirun in a distributed memory environment. In *Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)* (July 1998).

[83] MIROWSKI, M. The automatic implantable cardioverter-defibrillation: An overview. *Journal of the American College of Cardiology 6* (1985), 461–466.

[84] MUSSER, D., AND SAINI, A. *C++ Programming with the Standard Template Library*. Addison-Wesley, California, 1996.

[85] NIELSON, G. Challenges in visualization research. *IEEE Transactions on Visualization and Computer Graphics* (1996), 97–99.

[86] NIELSON, G., HAGEN, H., AND MULLER, H. *Scientific Visualization: Overviews, methodologies, and techniques*. IEEE Computer Society, 1997.

[87] OGI, T., AND HIROSE, M. Multisensory data sensualization based on human perception. In *Proceedings of the IEEE 1996 Virtual Reality Annual International Symposium* (Santa Clara, CA, 1995).

[88] OUSTERHOUT, J. *Tcl and the Tk Toolkit*. Addison-Wesley, New York, 1994.

[89] PARKER, S., BEAZLEY, D., AND JOHNSON, C. Computational steering software systems and strategies. *IEEE Computational Science and Engineering 4*, 4 (1997), 50–59.

[90] PARKER, S., AND JOHNSON, C. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95* (1995), IEEE Press.

[91] PARKER, S., MILLER, M., HANSEN, C., AND JOHNSON, C. An integrated problem solving environment: The SCIRun computational steering system. In *31st Hawaii International Conference on System Sciences (HICSS-31)* (1998).

[92] PARKER, S., WEINSTEIN, D., AND JOHNSON, C. The SCIRun computational steering software system. In *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen, Eds. Birkhauser Press, 1997, pp. 1–44.

[93] PATTANAIK, S., FERWARDA, J., FAIRCHILD, M., AND GREENBERG, D. A multiscale model of adaptation and spatial vision for realistic image display. In *ACM Siggraph '98 Conference Proceedings*, pp. 287–298.

[94] PLONSEY, R., AND BARR, R. C. *Bioelectricity*. Plenum Press, New York, 1988.

[95] Problem Solving Environments - Projects, Products, Applications and Tools: http://www.cs.purdue.edu/research/cse/pses/research.html.

[96] PURCIFUL, J. Three-dimensional widgets for scientific visualization and animation. Master's thesis, University of Utah, 1995.

[97] REED, D., ELFORD, C., MADHYASTHA, T., SMIRNI, E., AND LAMM, S. The next frontier: Interactive and closed loop performance steering. In *Proceedings of the 25th Annual Conference of International Conference on Parallel Processing* (1996).

[98] REED, D., SHIELDS, K., TAVERA, L., SCULLIN, W., AND ELFORD, C. Virtual reality and parallel systems performance analysis. *IEEE Computer* (Nov. 1995), 57–67.

[99] RIBARSKY, B., AND ET AL. Object-oriented, dataflow visualization systems—A paradigm shift? In *Proceedings of Visualization '92* (1992), IEEE Press, pp. 384–388.

[100] RICE, J., AND R.F., B. *Solving Elliptic Problems using ELLPACK*. Springer-Verlag, 1985.

[101] ROHLF, J., AND HELMAN, J. IRIS Performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)* (July 1994), A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 381–395. ISBN 0-89791-667-0.

[102] ROSENBLUM, L., EARNSHAW, R., ENCARNACAO, J., HAGEN, H., KAUFMAN,

A., KLIMENKO, S., NIELSON, G., POST, F., AND THALMANN, D., Eds. *Scientific Visualization Advanced and Challenges*. Academic Press, 1994.

[103] SALISBURY, K., BROCK, D., MASSIE, T., SWARUP, N., AND ZILLES, C. Haptic rendering: Programming touch interaction with virtual objects. *Symposium on Interactive 3D Graphics* (1995), 123–130.

[104] SCHMIDT, J. *Mesh generation with applications in computational electrophysiology*. PhD thesis, Duke University, Durham, NC, 1993.

[105] SCHMIDT, J., JOHNSON, C., EASON, J., AND MACLEOD, R. Applications of automatic mesh generation and adaptive methods in computational medicine. In *Modeling, Mesh Generation, and Adaptive Methods for Partial Differential Equations*, I. Babuska, J. Flaherty, W. Henshaw, J. Hopcroft, J. Oliger, and T. Tezduyar, Eds. Springer-Verlag, 1995, pp. 367–390.

[106] SCHMIDT, J., JOHNSON, C., AND MACLEOD, R. An interactive computer model for defibrillation device design. In *International Congress on Electrocardiology* (1995), ICE, pp. 160–161.

[107] SCHROEDER, W., MARTIN, K., AND LORENSEN, B. *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Prentice-Hall, 1996.

[108] SCHWEIGER, M., ZHUKOV, L., ARRIDGE, S., AND C., J. Optical tomography using the scirun simulation and visualization package: Preliminary results for three-dimensional geometries and parallel processing. *Optical Express: International Electronic Journal of Optics* (1999 (submitted)).

[109] SHEN, H., AND JOHNSON, C. Semi-automatic image segmentation: A bimodal thresholding approach. Tech. Rep. UUCS-94-019, Department of Computer Science, University of Utah, 1994.

[110] SHIRLEY, P. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illnois at Urbana-Champaign, Urbana, Illnois, 1991.

[111] SILBERSCHATZ, A., PETERSON, J., AND GALVIN, P. *Operating Systems Concepts*. Addison-Wesley, California, 1991.

[112] SILICON GRAPHICS, INC. *IRIS Performer Programmer's Guide*.

[113] SILICON GRAPHICS, INC. *Topics in IRIX Programming*.

[114] SONG, D., AND GOLIN, E. Fine-grain visualization algorithms in dataflow environments. In *IEEE Visualization '93* (1993), pp. 126–133.

[115] SPEARES, W., AND BERZINS, M. A 3d unstructured mesh adaptation algorithm for time dependent shock dominated problems. *International Journal of Numerical Methods in Fluids* (1997), 81–104.

[116] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley, Reading,

1991.

[117] SWANN, P., AND SEMWAL, S. Volume rendering of flow-visualization point data. In *Proceedings of Visualization '91* (1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 25–32.

[118] The UCL optical tomography software system (TOAST). http://www.medphys.ucl.ac.uk/toast/index.htm.

[119] TOMLIN, A., BERZINS, M., WARE, J., SMITH, J., AND PILLING, M. On the use of adaptive gridding methods for modelling chemical transport from multi-scale sources. *Atmospheric Environment 31*, 18, 2945–2959.

[120] TRELEAVEN, P., BROWNBRIDGE, D., AND HOPKINS, R. Data-driven and demand-driven computer architecture. *Computing Surveys* (March 1982), 93–143.

[121] VAN LIERE, R., AND VAN WIJK, J. CSE: A modular architecture for computational steering. In *Eurographics Workshop on Scientific Visualization '96*.

[122] VERWER, J. Gauss-Seidel iteration for stiff o.d.es from chemical kinetics. *SIAM J. Sci. Com*, 15:1243–1250.

[123] VETTER, J. Computational steering annotated bibliography. *Georgia Institute of Technology Technical Report* (1997).

[124] VETTER, J., AND SCHWAN, K. Progress: A toolkit for interactive program steering. In *Proceedings of the 24th Annual Conference of International Conference on Parallel Processing* (1995).

[125] VETTER, J., AND SCHWAN, K. Models for computational steering. In *Proceedings of the Third International Conference on Configurable Distributed Systems* (1996).

[126] VETTER, J., AND SCHWAN, K. High performance computational steering of physical simulations. In *Proceedings of the 11th International Parallel Processing Symposium* (Apr. 1997), Geneva, Switzerland.

[127] WATSON, D. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Computer Journal 24*, 2 (1981), 167–172.

[128] WEATHERILL, N., AND HASSAN, O. Efficient three-dimensional grid generation using the Delaunay triangulation. In *Proceedings of the 1st European CFD Conference* (1992), vol. 1.

[129] WEINSTEIN, D., POTTS, G., TUCKER, D., AND JOHNSON, C. The SCIRun inverse EEG pipeline - a modeling and simulation system for cortical mapping and source localization. In *First International Conference on Medical Image Computing and Computer-Assisted Intervention* (1998 (submitted)).

[130] WERNECKE, J. *The Inventor Mentor : Programming Object-Oriented 3D Graph-*

ics With Open Inventor. Addison-Wesley, 1994.

[131] WILLIAMS, C., RASURE, J., AND HANSEN, C. The state of the art of visual languages for visualization. In *IEEE Visualization '92* (1992), pp. 202–209.

[132] YU, F., AND JOHNSON, C. An automatic adaptive refinement and derefinement method for elliptic problems. *Proceedings of IMACS 3* (1994), 1555–1557.

[133] YU, F., LIVNAT, Y., AND JOHNSON, C. An automatic adaptive refinement and derefinement method for 3D elliptic problems. *Applied Numerical Mathematics* (1997 (to appear)).