BioPSE **User's Guide**

Version 1.0

Last update: March 23, 2001

Authors: Rob MacLeod (macleod@cvrti.utah.edu),
Ted Dustman (dustman@cvrti.utah.edu),
and the SCI Gang

Scientific and Computing Institute (SCI)
www.sci.utah.edu

Nora Eccles Harrison
Cardiovascular Research and Training Institute (CVRTI)
www.cvrti.utah.edu

Support for this project came from:
The NIH National Center for Research Resources (NCRR)

Contents

1 Introduction

This is the ***SCIRun* User's Guide**. It describes the purpose and use of the *SCIRun* problem solving environment. It is for those users who will be building and executing *networks* within the *SCIRun* environment.

Those who will be installing *SCIRun* should read the *SCIRun* Installation Guide[1]

1.1 Road Map

This document is organized into the following main sections:

Section 1 You are reading this section now.

Section 2 Introduces the concept of an integrated problem solving environment generally and describes how *SCIRun* embodies some of these ideas.

Section 3 Procedure for starting *SCIRun* and related information.

Section 4 Discusses tasks involved in building, editing, and executing networks.

Section 5 Describes the purpose and use of a visualization module called the *Viewer*. The *Viewer* is probably *SCIRun*'s most commonly used module.

1.2 Getting Help

1.2.1 In the Distribution

This document and others related to *SCIRun* can be found in your *SCIRun* distribution. Point your browser at `index.html` which is in your distribution's top level `doc` directory (*i.e.,* <top of distribution>/doc/index.html). Latex and postscript versions of some documents are also in the distribution's `doc` directory.

1.2.2 On the Web

This and other documents related to *SCIRun* can be found in the document[2] section of the *SCI*[3] web site.

Be sure to visit the main *SCI* web site for lots of other information related to *SCIRun* and the SCI Institute.

1.2.3 From the Mailing Lists

The *SCIRun* users mail list is a forum for discussing *SCIRun* related issues. To subscribe send mail to:

`Majordomo@cs.utah.edu`

with the following command in the body of your message:

`subscribe scirun-users`

1.3 Reporting Bugs

Please report bugs! To report a bug visit *SCIRun*'s bug database[4] web page.

Reporting bugs this way (rather than by way of the mailing list) ensures that they will be fixed in timely manner.

[1]On the web see http://www.sci.utah.edu/scirun/doc/InstallGuide/pseInstall.html. In the distribution see `doc/InstallGuide/pseInstall.html`

[2]http://www.sci.utah.edu/scirun/doc

[3]http://www.sci.utah.edu

[4]http://www.sci.utah.edu/bugs

2 Concepts

Read this section to learn about the design philosphy and goals of integrated problem solving environments generally and how *BioPSE* embodies some of these ideas.

2.1 Traditional problem solving methods

The traditional method for solving bioelectric field problems uses multiple, non-integrated computer programs. For example, a scientist using a computer simulation to examine the effect of electrode patch placement on transcardiac current density in the design of a cardiac implantable defibrillator[1] would require geometric modeling, numerical simulation, and scientific visualization tools to complete the task. The user might need one program to define the thoracic surfaces from medical images and another to create a discrete mesh of the volume contained within the surfaces[2]. Another application like Matlab compute a finite element simulation of the electric current distribution from the defibrillation electrodes through the thoracic volume[3]. Another approach might be to write a Fortran program using a public domain numerical library such as LAPACK . To see the output would require a scientific visualization package (such as those described in[4]). 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 could read—this might necessitate separate file format conversion utilities. To find the optimal location, shape, and size parameters for the defibrillating electrode, the scientist would have to go back to the geometric modeling package, change the necessary parameters, manually re-run 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 could 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 could 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 inter-package execution is *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 most available computer programs execution cannot be automated. That is, the package cannot be run without regular user intervention during execution. This constraint makes it difficult or impossible to run multiple computational jobs automatically, leaving the user with the task of manually initiating and controlling each step of the process.

2.2 Integrated problem solving and computational steering

The goal of integrated problem solving environments—and specifically of *SCIRun* and *BioPSE*—is to incorporate and integrate all the steps described in the previous example as components in a single, unified, extensible problem solving environment (PSE). The functionality that will result includes the ability to manage each step in a sequential computing process, and to create batch processes that execute repeated simulations. However, the functionality that sets *SCIRun* and *BioPSE* apart from most integrated software environments is the ability to intervene and control execution anywhere in the chain at any time during its execution. This ability to control a computer program during execution is termed *computational steering.*

To provide a non-technical analogy, adding computational steering to a software environment is similar to adding the ability to occasionally switch tracks to train travel. A train passenger can get on the train and automatically get to a new destination, leaving all the details of the individual actions to the rail system machinery and staff. But the route and the destination are fixed. Steering would permit each passenger to request that the train take a new route, with different stops, and even a different destination, and be able to make these decisions at any time during the trip. In the more rigorous example of the defibrillation simulation, computational steering allows a scientist to interactively change parameters and settings as the simulation executes, both as a single run or in batch mode. Steering interventions might include adjusting electrode locations to stay within anatomically reasonable bounds or refining the geometric model resolution in order to balance accuracy and execution time.

To achieve integration within the elements of *SCIRun* and *BioPSE*, data flows directly from one processing step to the next, without ever being diverted to a disk file or leaving the program. Output from any step are available as inputs to dependent steps. The underlying paradigm of *SCIRun* is of data flowing between modules that each perform some operation. Integration between modules guarantees that upon completion of their tasks, upstream modules pass their data to downstream modules, thereby forcing the downstream modules to execute in response. In our example, this means that the scientist may alter electrode locations at any time, thus initiating a sequence of all the necessary steps to recompute the simulation with the new configuration. The modification of the geometric model, finite element calculation, and visualization all proceed automatically and in the proper sequence, all managed by *SCIRun*. This combination of steering and component integration allows the scientist to spontaneously explore a problem.

While computational steering is still a young field in computer science, there are a number of examples of such systems (besides *SCIRun*) described in the literature. Burnett[5], and Vetter and Schwan[6] give overviews of existing computational steering system. Several notable examples include CUMULVS[7,8], Progress[9], and Magellan[10] .

2.2.1 *SCIRun* versus *BioPSE*

It is important to understand the hierarchy of the problem solving software included in this package. From a historical perspective, *SCIRun* was the original implementation of the concept of dataflow in a computational framework[11−16]. We have now generalized and greatly expanded the *SCIRun* software so that now, *SCIRun* is the overarching term for any of the integrated problem solving applications built on the same foundation. *BioPSE* is then one of these specific applications, in this case for solving bioelectric field problems. Another example of a *SCIRun* applications package is Uintah[17], which focuses on the simulation of fire and combustion.

In this manual, we will try and be consistent about the usage of the two terms, *SCIRun* and *BioPSE*. *SCIRun* will typically refer to some feature that is common to the core functionality of the system and this is common to all of the problem solving environment applications packages. *BioPSE* will refer to specific elements of the bioelectric field problem solving environment.

2.3 *SCIRun* modules and networks

The functional unit of a dataflow environment is the *module* . Figure 1 contains a generic *SCIRun* module, with a User Interface (UI) button for graphically accessing the module's parameters, and input and output ports for receiving and sending data, respectively. On the right is a simple example of a dataflow network. Data passes through the output port of the top module, through the data pipe, and into the input port of the bottom module. The User Interface enables the selection of a desired isochrone surface.

Modules may contain other elements, but they will all have at least one input or one output port. Examples of a module with only an output port would be file readers. The *Viewer* module contains only input ports because it receives information and the output is a visualization. For a more detailed description of modules and how to control them, see Section 4. The most important goal at this stage is to appreciate the concept of a module and the dataflow that links one module to another.

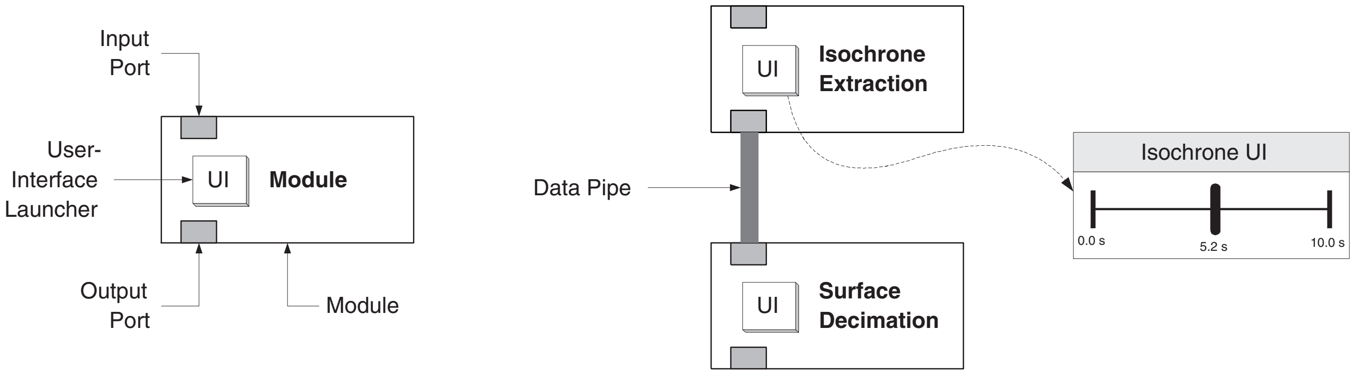


Figure 1: Example of a *SCIRun* module

A network diagram describes the way data flows through *SCIRun* and is the main means of user interaction with the overall action of the program. The network essentially defines the basic function of the program; without a network, *SCIRun* and *BioPSE* are just a set of tools sitting in a chest. By joining the modules into a network, the tools become a functioning program that does whatever the network tells it to. Once again, the important thing to appreciate is what the network is conceptually; for the details, see Section 4.

2.4 Links to third party software

SCIRun works together with software from third party sources in several ways. If you have installed *SCIRun* and *BioPSE* yourself, you will have seen the need for some third party software libraries that are required for basic functionality. This type of third party software is largely invisible to the user of *SCIRun* or *BioPSE*, at least until something breaks... Perhaps the most evident is the TCL/Tk library, which *SCIRun* uses to create the network editor and the icons for the modules and data pipes. In fact, *SCIRun* is actually a large TCL script that calls a lot of specialized C++ code to do all the hard work.

Far more interesting is the interaction between programs you may have that are written in FORTRAN or Matlab and *SCIRun*. One of the goals of the *BioPSE* project was to develop support for such external code, including FORTRAN, C, Matlab, and IDL. This first version of *BioPSE* has support for Matlab and one example of wrapping existing FORTRAN code into a *SCIRun* module.

2.4.1 *SCIRun*/Matlab interface

The interface (based on Berkeley sockets) between *SCIRun* and Matlab provides a pathway to send matrix data objects from *SCIRun* to Matlab and then accept the result of some Matlab computations. At present, this arrangement requires that a Matlab script exist that will perform the desired operations. *SCIRun* sends the input data to an existing process running Matlab, which serves as a compute engine, performs the steps described in the script, and then returns data to *SCIRun* for further processing or display. The Matlab process can even run on a separate computer connected via a network, which helps to distribute the load as well as to resolve potential licensing conflicts with Matlab.

The underlying mechanism for this communication is a socket interface consisting of two SCIRun Modules, `MatrixSend` and `MatrixReceive`, and a Matlab "transport" routine. Both the *SCIRun* and the Matlab process know about each other's whereabouts (in the form hostname:port) and use a client-to-client communication model, so that synchronization between processes is manual. For example, the *SCIRun* `MatrixSend` module sends the matrix to a socket at which a Matlab script is listening. The script then receives the matrix, performs the calculations in Matlab and sends the results to a socket where `MatrixReceive` module is listening. *SCIRun*then carries out further calculations and display of the results.

2.5 Extensibility

SCIRun is an extensible problem solving environment. This is true in the sense that no one by the user really limits the different ways of connecting modules and creating new applications. Ii is also more generally true in that *SCIRun* is designed to have users who can program create their own new modules. To support creating new modules, we have developed the *Module maker* application, described in The Module maker manual.

With the release of *SCIRun*, we anticipate that users all over the world will create new modules and we will encourage them to contribute modules to a repository on the *BioPSE* web site (www.sci.utah.edu/ncrr/software/biopse.html. We will review submissions to this collection of modules and adopt and then test generally useful ones to include in subsequent releases of *BioPSE*. Future releases will also include more extensive tools for both building modules and wrapping existing codes within *SCIRun* module wrappers to maximize your intellectual investment in legacy code.

2.6 Detachable interface

A strategy seen in a few PSEs currently under development and an integral part of *SCIRun* is the idea of a detachable interface. Rather than having a fixed link between a user interface and the underlying application, a detachable interface can be separated from the application, in which case the application becomes unsteerable. The advantage, however, lies in the ability to now re-attach an interface to a running application and thus re-enable steering. The re-attached interface need not be from the same physical computer from which the application was started, instead it may come from a remote computer via a network, or from another user on the same computer. An additional strength of this approach is that the re-attached interface may not be an interactive widget at all, but a set of commands executed from a script. This permits complete remote control of the application and thus is a flexible blend of interactive and almost unlimited batch control. CUMULVS is one example of such a system and uses its detachable interface to allow collaboration between scientists working remotely, with viewing systems that can attach or detach from the running simulation.

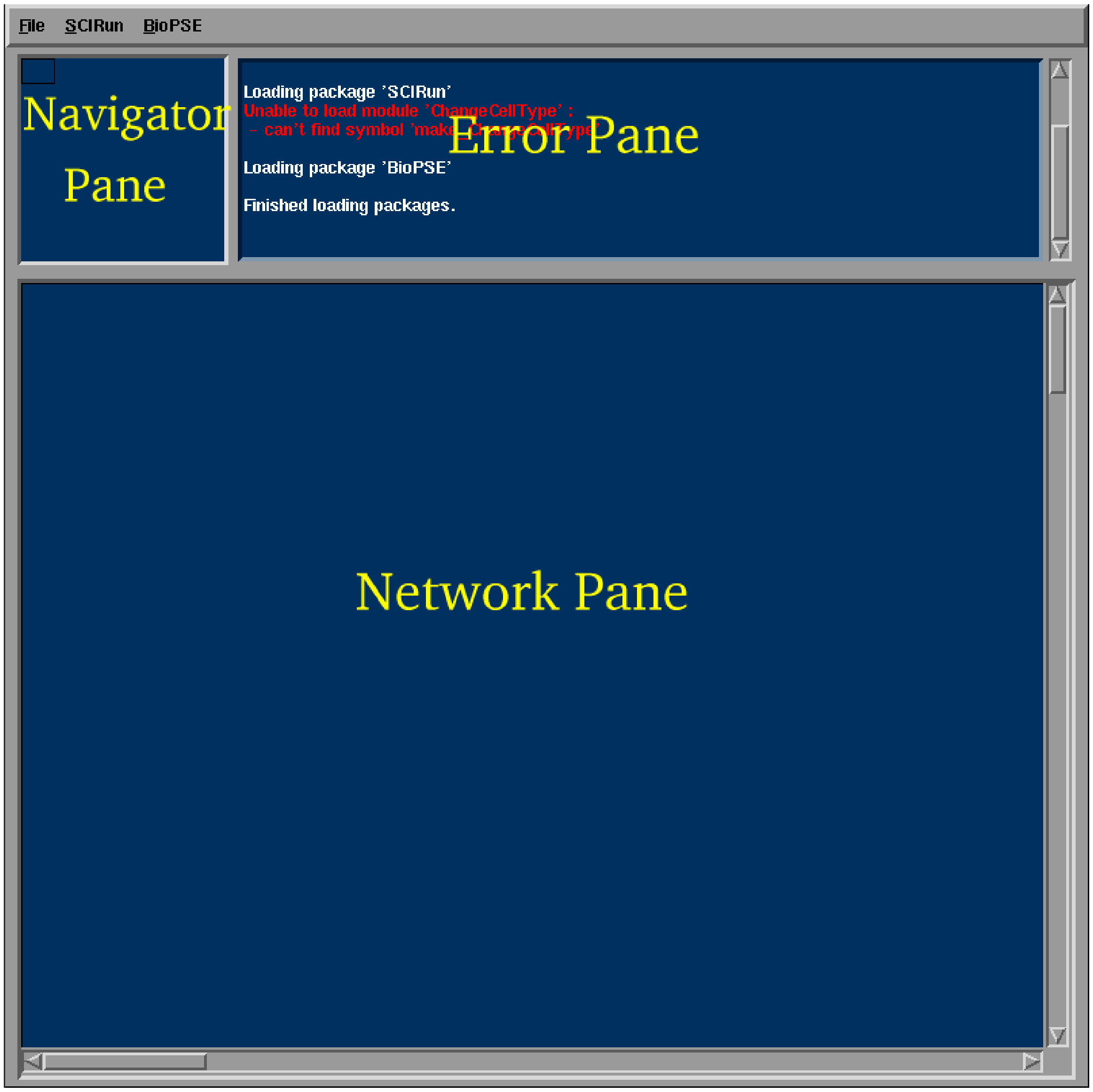
The first release of *SCIRun* will not include any of the detachable interface support but we plan to add this within six months of the first release.

3 Starting *SCIRun*

3.1 The `scirun` Command

Start *SCIRun* by typing **`scirun`** in a terminal (*e.g.*,`xterm`) window. *Note: Don't start* SCIRun *in the background,* i.e.,*don't type* ***`scirun &`***.

The **`scirun`** command is located in the **`src`** directory of the ***SCIRun*** install directory. The person who installed *SCIRun* can locate this command for you.



```
Loading package 'SCIRun'
Unable to load module 'ChangeCellType' :
 - can't find symbol 'make_ChangeCellType'

Loading package 'BioPSE'

Finished loading packages.
```

Figure 2: *SCIRun* Main Window

Typing **`scirun`** with no arguments starts up *SCIRun* with a blank *SCIRun* window as shown in Figure 2. The main features of this window are discussed in Section 3.1.1.

The **`scirun`** command may take 1 argument which is the name of a *SCIRun network* file (these files have a **`.net`** extension). These files hold previously defined *SCIRun* networks. *SCIRun* will load the specified network. Network files will be discussed in a later section.

SCIRun may encounter errors during start up. These will be displayed in *SCIRun*'s error message pane (see Figure 2) (These errors should be reported the *SCIRun* development team. See Section 1.3 these errors. this window shou

3.1.1 Anatomy of the Main Window

The *SCIRun* main window consists of 4 main components (see Figure 2):

Menu Bar The menu bar is used to load networks, save networks, quit *SCIRun*, create network modules, and perform other tasks. The menu bar consists of the following menu items:

File The **File** menu contains the following items:

Save Saves the current network to a file.

Load Loads a network from a file.

New This submenu contains items of interest to developers only.

Add Info Use this item to add network specific notes to the current network. Notes should be used to document the purpose of the network.

Quit Quits *SCIRun*.

SCIRun The **SCIRun** menu is used to create modules (from the *SCIRun* package) for use in the network pane. This menu is composed of submenus. Each submenu corresponds to a *category* within the *SCIRun* package. A category is group of related modules. Each menu item in a category submenu creates a specific module and places it in the network pane. The network pane's popup menu (activated by clicking the right most mouse button when the mouse pointer is in the network pane) also provides access to the ***SCIRun*** and ***BioPSE*** (and possibly other) package menus. An overview of the contents of the *SCIRun* package is given in Section **??**.

BioPSE The **BioPSE** menu is used to create modules (from the *BioPSE*package) for use in the network pane. It consists of category submenus and module menu items. An overview of the contents of the *SCIRun* package is given in Section **??**.

Other Package Menus There may be other package menus if other packages have been installed. They too will consist of category submenus and module menu items.

Navigator Pane The Navigator Pane is located in the upper left corner of the main window (see Figure 2). It is used to navigate complex networks. The use of the Navigator Pane will be described in Section 4.14.

Error Pane Errors during program startup are displayed in the Error Pane. It is located in the upper right corner of the main window(see Figure 2). Errors on startup may mean that *SCIRun* has been installed incorrectly or has been installed from a buggy distribution. Please (see Section 1.3) these errors.

Network Pane The Network Pane occupies the bottom of the main window(see Figure 2). It is used to build and execute networks. Section 4 discusses the use of this pane.

3.2 The Terminal Window

After starting, *SCIRun* will also run a shell-like application in the terminal window called the SCIRun *shell*. The *SCIRun* shell displays the prompt `scirun>`. This program is actually a modified *Tool Command Language* (TCL) shell program and it is possible to type in TCL'ish *SCIRun* commands at the prompt. The use of this program will be described in Section 4.15.

3.3 SCIRUN_DATA

The environment variable `SCIRUN_DATA` specifies the default directory of *SCIRun* data files. *SCIRun*will first look in this directory to find data and network files. It affects the behavior of file browsing dialogs – they will prompt for a file within the `SCIRUN_DATA` directory (of course you may have the dialog look elsewhere).

3.4 Stopping

Quit *SCIRun* by selecting the **Quit** item from the **File** menu

Warning: Don't press `control-c` *to exit* SCIRun. *Doing this will drop you into a debugger which is probably not what you want to do.*

4 Working with Networks

This section describes how to create, save, open, execute, and edit networks.

When started with no arguments the `scirun` command will create a main window with a blank network pane. The goal then is to create and connect modules to form a network.

4.1 Creating a Module

To create a module select its name from one of the package (*e.g., SCIRun*) menus' category submenus. The package menus may be accessed from the main window's menu bar and from the network pane's popup menu.

You may activate the network pane's popup menu by clicking mouse button 3 while the mouse pointer is in the network pane (but not over a module or connection).

The popup menu contains a list of category submenus from the *SCIRun* package and other installed packages. And each category submenu provides access to the modules within the category.

After creating a module its graphic "front end" will be created and placed in the network pane.

4.1.1 Anatomy of a Module

All modules are represented similarly by a graphic within the network pane. See Figure 3. This graphical "front end" is the same for all modules and consists of the following components:

Name The module's name.

Input ports Zero or more input ports located on the top of the module. Each port corresponds to a data type and each data type has a unique color. Table 1 maps port colors to data types. Input ports connect to other modules' output ports. Connections can only be made between ports of the same type.

Output ports Zero or more output ports located on the bottom of the module. Output ports connect to other modules' input ports. Every module has, of course, at least 1 input or 1 output port.

UI button Pressing the **UI** button displays the module's control dialog. Some modules have no such dialog. Some have very simple dialogs. Some have very complex dialogs which allow elaborate control over the module. Figure 4 shows the control dialog for `Show Field` module.

Progress bar Shows the module's progress.

Timer Displays the amount of CPU time the module has consumed. Located to the left of the progress bar.

Popup Menu Pressing mouse button 3 while the mouse pointer is over a module gives access to the module's popup menu. The popup menu gives access to the following items:

- **::Package_Category_Name_Instance** This item is a label (not a selectable item). It provides the module's name and the category and package to which to module belongs. The Instance part is a unique number which distinguishes multiple instances of the same module.
- **Execute** Executes the network.
- **Notes** This item displays the module's note pad. Use the note pad to document the purpose of the module in the current network.
- **Destroy** Destroys the module.
- **Show Log** Displays the module's message log. Most modules will write messages to their log during the course of their execution.
- **Show Status** This item is a toggle button which turns off/on the display of the progress indicator. Turning off the progress indicator can help speed up the execution of complex networks.

Data Type	**Port Color**
Field	Yellow
Field Set	Green
Matricies	Blue
Geometric Objects	Pink
Color Maps	Purple
Camera Path	Brown

Table 1: Data Types and their Port Colors

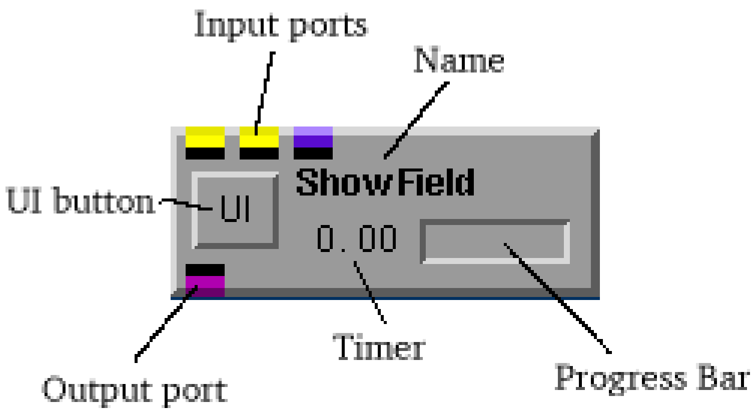
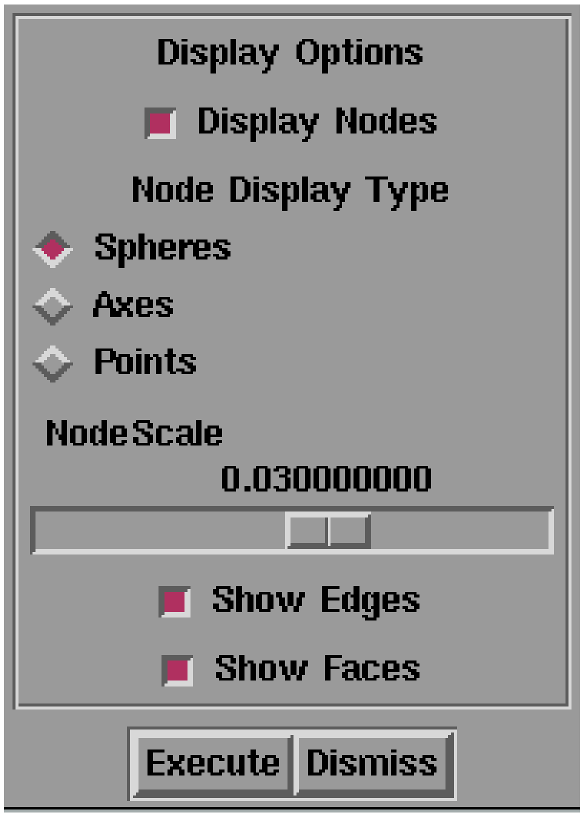


Figure 3: Module Graphic (`Show Field` Module)

4.2 Creating a Connection

Mouse button 2 is used to connect the output (input) port of one module to the input (output) port of another module.



Display Options

[✓ Display Nodes]

Node Display Type

[✓ Spheres]

Axes

Points

NodeScale 0.030000000

[✓ Show Edges]

[✓ Show Faces]

Figure 4: Module Control Dialog (`Show Field` Module)

Position the mouse pointer over a module's input (output) port. Then press button 2 and drag the the mouse pointer towards another module's output (input) port.

When button 2 is first pressed the program shows all valid connections as black lines. It also draws one red colored connection which is the connection that will be made if you stop the drag by releasing button 2.

Make the final connection by releasing button 2 when the pointer is over the desired destination port or when the red colored connection is the desired one. The connection will be drawn using an appropriate data type color.

You may connect a module's output port to the input ports of 1 or more modules.

4.3 Setting Module Properties

To change a module's properties click its **UI** button. This will display the module's control dialog. Use this dialog to change the module's properties. Each module's reference documentation explains the use of its control dialog. Figure 4 shows the control dialog for the `Show Field`.

4.4 Executing a Network

"Network Execution" means that 1 or more modules must be executed in a coordinated fashion. The coordinated execution of modules is managed by *SCIRun*'s *scheduler*.

4.4.1 The Basics

The scheduler is invoked when an event occurs that *triggers* a module's execution. The scheduler creates a list of all modules that must executed in coordination with the triggered module. Modules *upstream* (directly or indirectly) from the triggered module will be put on the execution list if they have not previously executed. And all modules *downstream* from the triggered module will be put on the list. Once the scheduler determines which modules must be executed, it executes them.

Network execution is mostly transparent. That is, the events that trigger module execution will usually be generated automatically while you work (*e.g.,*by changing a module's property). Sometimes though you must manually generate a triggering event by choosing the **Execute** item from a module's popup menu.

4.4.2 Details

Each module executes in its own thread and blocks (waits) until its upstream modules can supply it with data. After a module completes its computation it sends its results to its downstream modules. This completes a module's execution cycle. Its next chance to recieve data from its input ports and send data to its output ports won't arise until some event causes it to be put on the scheduler's execution list again.

This behavior prevents modules from computing in an iterative fashion, sending intermediate results to their downstream modules. This is because downstream modules can't recieve these results until they are in their execution cycle. They would need to be executed each time the upstream module posts an intermediate result.

4.4.3 Intermediate Results

Some modules are designed to be used in an interative fashion. They send data to their output ports in a special way. They use a method called `send_intermediate` to send the results of each iteration. When this method is used the scheduler (re)executes downstream modules each time the upstream module posts its next result. Downstream modules are therefore able to receive the results of each iteration as soon as the upstream module sends them.

Currently module `SolveMatrix` (from the (*SCIRun*) package and `Math` category) is the only module that computes iteratively using the `send_intermediate` method.

4.4.4 Feedback Loops

Some modules that are designed to be used in feedback loops (and if fact, they can only be used this way). Their output ports may be connected directly or indirectly to their input ports. These modules also use the `send_intermediate` method.

The list of feedback modules is `DipoleSearch` and `ConductivitySearch` from the `Inverse` category of the `BioPSE` package and `BuildLeadField` from the `LeadField` category of the `BioPSE` package.

4.5 Deleting a Connection

Delete a connection by pressing button 3 while the pointer is over a connection.

4.6 Selecting Modules

A few operations (see Section 4.7 and Section 4.8) act on a group of modules. There are 2 ways to create a group of modules.

1. Press mouse button 2 while the pointer is in the network pane but not over a module or a connection. Then drag the mouse. This will draw a box outline. Modules intersecting the box will be part of the group. Release button 2 to complete your selection.

2. Select the first module by clicking mouse button 2 while the pointer is over a module. Then add modules to the group by pressing (and holding down) the `control` key while you click on modules with button 2.

Selected modules will be drawn in a slightly darker shade of grey.

You may mix selection methods – you may always add modules the group by pressing the control key while making selections using the above methods. If you don't press the control key then the previous group will be forgotton and a new one will be created.

4.7 Moving Module(s)

Modules may be moved around in the network pane. To move a module simply press button 1 while the pointer is over a module and drag the module to its new location.

Multiple modules may be moved at one time. To do this first select (see Section 4.6) 1 or more modules. Then press button 1 while the pointer is over any one of the modules in the selected group and then drag the modules to their new location.

4.8 Destroying Module(s)

Delete a module by selecting the **Destroy** menu item from the module's popup menu.

Multiple modules may be deleted at one time. To do this first select (see Section 4.6) 1 or more modules. Then choose, from the popup menu of any one of the modules in the selected group, the **Destroy Selected** item.

4.9 Documenting a Module

It is often useful to document the purpose of a module within a network. Each module maintains a note pad for this purpose. You may edit the module's note pad by selecting **Notes** item from the module's popup menu. This will display the module's note pad editor. This simple editor allows you to write notes or otherwise document the use of the module within the context of its network.

4.10 Viewing a Module's Log

Each module supports a message log. The module may write error messages or other types of messages to its log. To view this log select the **Show Log** item from the module's popup menu.

4.11 Documenting a Network

It is useful to document the purpose of a network. You may use a network's note pad for this purpose. To edit the network's note pad select the **Add Info** item from the main window's **File** menu. This will display the network's note pad editor. This simple editor allows you to write notes or otherwise document the purpose and use of the network.

4.12 Saving a Network

SCIRun can save networks to files. Network files have an extension of `.net` (although in the past they have also had .sr and .uin extensions).

To save a network select the **Save** item from the main window's **File** menu. A file browser dialog will prompt you for the name (and location) of the network file.

Network files are actually *Tool Command Language* (TCL) scripts and are editable (although this topic is beyond the scope of this guide).

4.13 Opening a Network

To open a network file select the **Open** item from the main window's **File** menu. A file browser dialog will prompt you for the name (and location) of the network file.

Note that opening a network file merges the network in the file with the network in the network pane. There is currently no way to "close" a network and then open a new one – you must quit and restart *SCIRun* to do this.

4.14 Navigating a Network

A complex network may not be entirely visible in the network pane. You may use the network pane's scroll bars to bring other parts of a network into view. Or you may use the navigation tool to view complex networks.

The navigation pane shows the entire "network world." The small rectangular region (outlined in black) within the navigation pane is the network navigation tool and it is a window on the network world. The position of the navigation tool determines the part of the network that is visible in the network pane. To view other parts of the network, press button 1 while the pointer is anywhere in the navigation pane – this will move the tool to the location of the pointer – then drag the tool to the new location.

4.15 The *SCIRun* Shell

After starting, *SCIRun* will run a shell-like application in the terminal window called the SCIRun *shell*. It displays the prompt `scirun>` in the terminal window. This program is a *Tool Command Language* (TCL) shell program that has been extended with *SCIRun* specific commands.

It possible to type TCL *SCIRun* commands at the prompt. For instance, to load a network file you may type `source` <***network file name***>. This has the same effect as the **File** menu's **Open** command.

5 Visualization with the *Viewer*

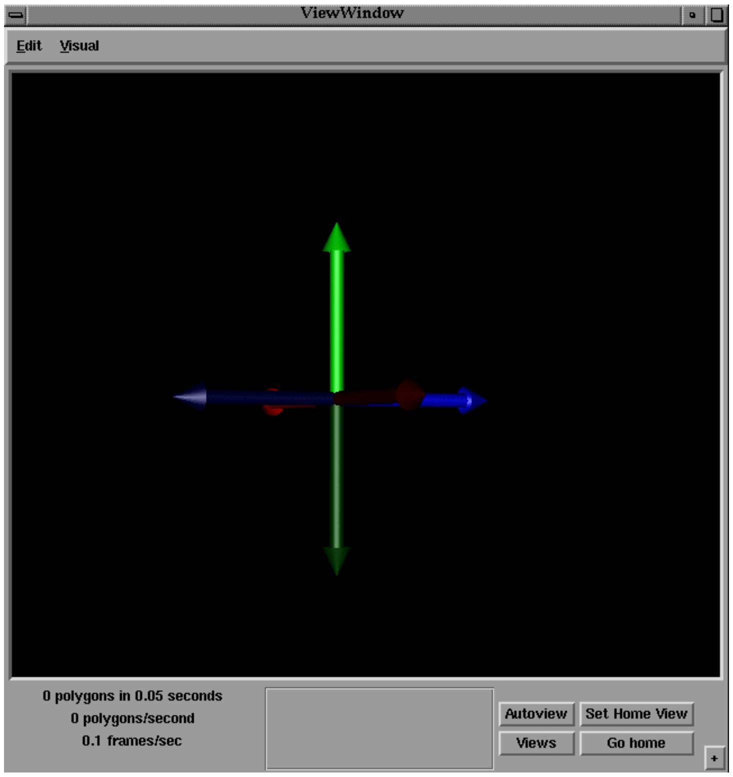
This section describes perhaps the most frequently used module of *SCIRun*, the *Viewer*, which has the task of displaying interactive graphical output to the computer screen. You will use the *Viewer* any time you wish to see a geometry, or spatial data. More important for the computational steering (described in Section 2.2), is that the *Viewer* provides access to many simulation parameters and controls and thus indirectly initiates new iterations of the simulation steps.

We begin with an overview of the *Viewer* window and its controls, then describe in detail all the options and variations.

5.1 Anatomy of the *Viewer* window

The *Viewer* window contains two main areas, the upper portion, called the *Graphics* window, which displays the graphics, and the lower portion, where most of the control buttons are. Figure 5 contains an example of a *Viewer* window. In the *Graphics* window, control is mostly by means of the mouse, mouse buttons, and various modifier keys (shift/control/alt). In the lower window are a lot of buttons and sliders, the function of which will become clear when you read this manual.

First, try out the controls for the *Graphics* window by moving the mouse to the center of the viewer window and clicking and holding the left button and then dragging the mouse. The objects should translate along with the mouse. Do the same operation with the middle mouse button and the objects will rotate around a point in the center of the display. The right mouse button controls the scale of the display, zooming in when the mouse moves downward or to the right. See Section 5.2 for all the gory details on mouse control.



0 polygons in 0.05 seconds
0 polygons/second
0.1 frames/sec

Figure 5: The default *Viewer* window in *SCIRun*

The control visible along the bottom of the *Viewer* window set some basic configurations as follows:

Autoview: restores the display back to a default condition, very useful when some combination of settings results in the objects disappearing from the view window.

Set Home View: captures the setting of the current view so you can return to it later by clicking the "Go home" button.

Go home: restore the current home view.

Views: lists a number of standard viewing angles and orientations. The view directions align with the Cartesian axes of the objects and the "Up vector" choice sets the orientation of the objects when viewed along the selected axis.

In the left corner of the control panel of the *Viewer* window are performance indicators that document the current rendering speed for the display. The better the graphics performance of the workstation you have, the higher the drawing rate.

In the lower right corner of the *Viewer* window is a small plus sign ("+"). Clicking on this reveals the extended control panel with controls that we will describe in detail in Section 5.3.

5.1.1 Menus

At the top of the *Viewer* window are two pull-down menus.

Edit: provides access to controls for the background color for the window, as well as the clipping planes (requires the "Use Clip" control to be selected in the extended controls described in Section 5.3).

Visual: allows you to select between different graphics hardware settings that are available on your workstation. The list is ordered heuristically from most to least useful.

5.2 Mouse control in the *Viewer* window

The mouse controls within *SCIRun* are extensive and flexible. The resulting action depends on the choice of mouse button, any simultaneous control keys, and the way the mouse moves. The description in Tables 2 and 3 below may seem overly complicated at first, but with a little playing, it becomes intuitive (another way of saying you will learn it if you use it enough).

Mouse Controls		
Control Key	Button	Action
None	Left	Translate scene
	Middle	Rotate scene about its center on an arc ball that surrounds it; rotation direction is a function of the initial mouse location so try different sites and note the response.
	Right	Zoom or scale scene (downwards and to the right increases size, upwards or to the left decreases size)
Shift	Left	Select and move a widget in the display
	Middle	Toggle through the modes for a widget
	Right	Pop up a widget information window
Control	Left	Translate in the Z-direction, *i.e.,* zoom in and out of the screen (down moves closer, up further away). Moving left and right increases the "throttle" of the Z-direction motion. If the cursor is over a point on an object when clicked, this point becomes the center of the screen for translation.
	Middle	Rotate the camera view about the eye point (using arcball motion).
	Right	Unicam movement (see next table)

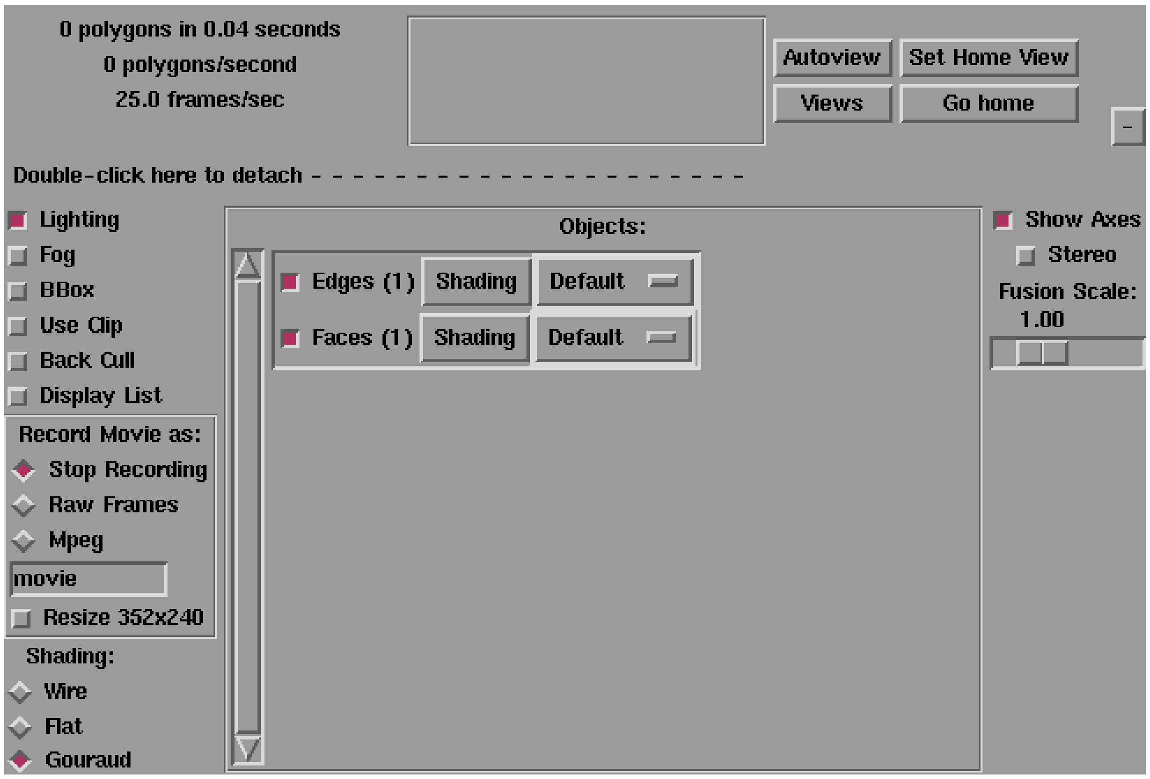
Table 2: Mouse controls for the *Viewer*

5.3 Extended control window

Click on the "+" sign in the lower right corner of the default *Viewer* window, and the window expands to reveal an extended panel of control buttons, as shown in Figure 6. Click on the "-" sign that now replaces the "+" and this extended panel disappears again. Here we describe the control options available in the extended control window.

Unicam movement (Control key and right mouse button		
Initial mouse location	Action	
Near edge of display	Rotate objects on the arc ball	
Near the objects	Following behavior:	
	Initial mouse movement	Action
	Horizontal	Pan objects
	Vertical	Zoom and pan: down = zoom in, up = zoom out, left and right= pan left and right)
	None	Set rotation point for subsequent arc ball rotation.

Table 3: Autocam mouse controls in the *Viewer*



0 polygons in 0.04 seconds
0 polygons/second
25.0 frames/sec

[✓ Lighting]
Fog
BBox
Use Clip
Back Cull
Display List

Record Movie as: [✓ Stop Recording] Raw Frames Mpeg
movie
Resize 352x240

Shading: Wire Flat [✓ Gouraud]

[✓ Edges (1)] Shading Default
[✓ Faces (1)] Shading Default

[✓ Show Axes]
Stereo
Fusion Scale: 1.00

Figure 6: The lower portion of extended *Viewer* window in *SCIRun*

5.3.1 Object selector

The lower portion of the extended *Viewer* window is divided into three columns and the middle of these contains a list of all the objects in the display. If the list becomes long enough, a scroll bar on the left hand side controls which are visible. For each entry in the list, we have the following controls, reading from left to right:

- At the left end of each of the objects in the list is box that displays red when that object is selected. The *Viewer* window only displays those objects that are selected.
- Next comes the name of the object.
- The $\boxed{\text{Shading}}$ control box that comes next determines the shading options that will be used for rendering the object. Options include: Lighting, BBox, Fog, Use Clip, Back Cull, and Display List (for descriptions, see Section 5.3.2).
- As the right end of each entry is the lighting control, initially marked $\boxed{\text{Default}}$. In the Default setting, the common rendering controls described in Section 5.3.2 below apply. Clicking this box reveals a set of options that will apply only to this object that include Wire, Flat and Gouraud.

5.3.2 Rendering controls

The left column of the extended *Viewer* window contains controls that apply to all of the selected objects with "Default" lighting selected. Those without the Default setting will use their own, object specific settings, as described in the previous section. The lighting and shading options available are:

Lighting: toggles whether or not the *Viewer* applies lighting to the display. Objects without lighting have a constant color.

Fog: draws objects with variable intensity based on their distance from the user, also known as "depth cueing". Close objects appear brighter while more remote objects fade gradually into the background as a function of distance from the front.

BBox: toggles whether the *Viewer* draws the selected objects in full detail or as a simple bounding box.

Use Clip: applies up to six clipping planes to the display. To control the clipping plane locations, use the "Edit -¿ Clipping Planes" menu at the top.

Back Cull: displays only the forward facing facets of any surface objects in the display.

Display List: cache the list of objects to be displayed; this option accelerates rendering when the content of the display does not change.

Shading: selects the type of shading for objects from the following options:

- **Wire:** show only the wire mesh of objects.
- **Flat:** draw each facet with a constant color.
- **Gouraud:** linearly interpolate the color across facets.

The right hand column of the extended *Viewer* window contains controls for displaying the axes and creating stereoscopic rendering.

Stereo viewing: requires hardware LCD glasses synchronized with the display so that visibility for each eye coincides with the display of the appropriate view. The "Fusion Scale" control provides a means of setting the eye separation and thus setting the view that is most suited to facial anatomy and distance from the screen.

5.3.3 Making movies

The *Viewer* window in *SCIRun* has simple controls for capturing sequences of images into animations or movies. Here we describe how this works.

In the left column of the extended *Viewer* window are controls for selecting movie type and then initiating and stopping the acquisition of individual frames in the movie.

SCIRun sends a frame to the movie after each "redraw" operation, *i.e.,* each time anything moves in the display or any visualization parameter changes. If the MPEG package is available (See the Installation Manual for details) then an option will be available for saving the animations as MPEG movies.

There is also a button that forces the size of the graphical window to be 352x240 pixels in size, which is a standard format well suited to MPEG.

5.4 Control widgets

While the mouse controls in Section5.2 describe many ways to interact with the contents of the *Viewer*, SR also supports some powerful display widgets. Examples of widgets capabilities include managing cutting surfaces colored according to the local data values, displaying streamlines in vector fields, or selecting sub-volumes within the display area for further manipulation.

We have tried to make interacting with these widgets as consistent as possible so that, for example, controlling parameters is usually by clicking and dragging on either a cylindrical "collar" or a sphere element of the widget. The original design of these modules was by James Purcifal Note that a single widget may have more than one purpose depending on the context in which it exists.

In this section, we describe the widgets available within *SCIRun* and *BioPSE*. The same widget may, for example, select a clipping or a display plane through a three-dimensional object but may also set the seed points for a streamline module.

5.4.1 Gauge Widget

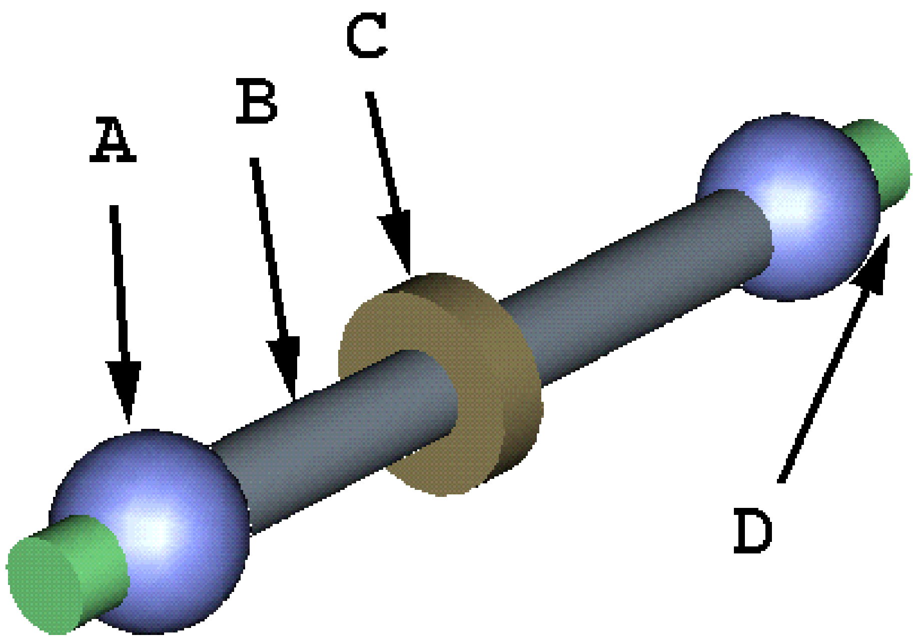


Figure 7: The gauge widget for setting location and density of seed points

Appearance: The Gauge Widget consists of two spheres (A) connected by a cylinder (B) with a small slider collar (C) on the cylinder. There are also small resize cylinders extending from the spheres (D).

Purpose: The primary use of the Gauge Widget is to set the location and density of streamlines emerging from the long cylinder. It may also be used as a more general purpose three-dimensional slider or as a source for a stream surface.

Controls: Clicking and dragging either sphere causes the entire widget to move in space, rotating about the other sphere and following along behind the selected sphere. Dragging either of the resize cylinders cases the size of the widget to change and dragging any point on the main cylinder moves the whole widget without any change in orientation. Dragging the slider collar changes the associated value, typically the density of seed points for a streamline source.

5.4.2 Frame Widget

Appearance: The Frame Widget consists of four cylinders connected in a rectangle. In the middle of each of the cylinders there is a sphere (B), from which a resize cylinder extends (C).

Purpose: The primary uses of the Frame Widget is for image plane definition, for defining stream volumes, and as a "tie dye" as with the Ring Widget described in Section 5.4.4.

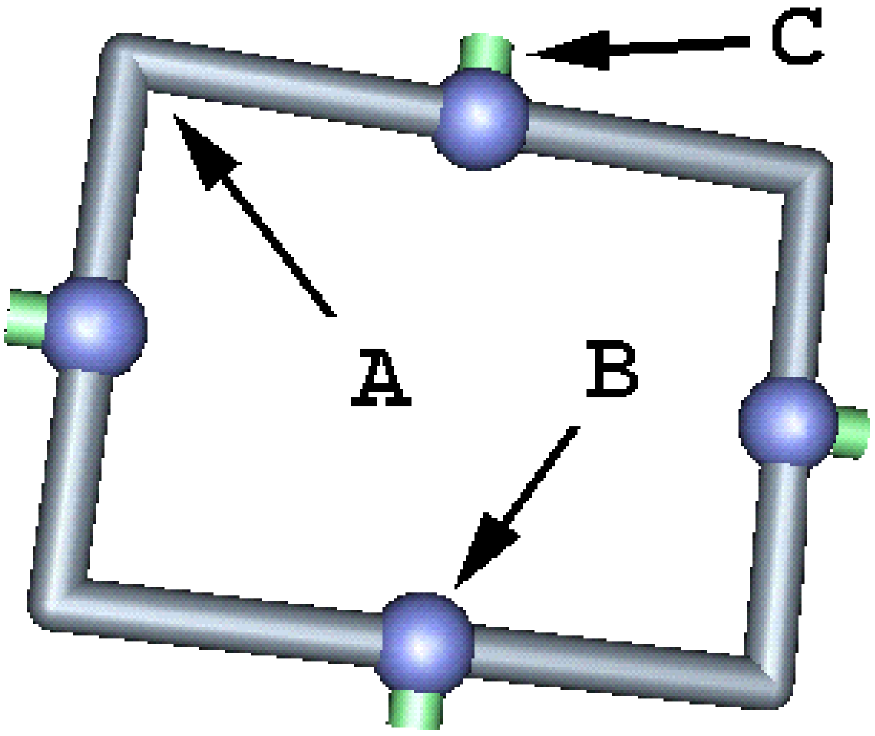


Figure 8: The frame widget for selecting cutting/projection planes

Controls: Clicking and ragging a sphere on the widget will cause the widget to rotate about it center; dragging on a resize cylinder will move the associated edge and this extend or contract the rectangle. Dragging any of the cylinder will drag the entire widget through space.

5.4.3 Box Widget

Appearance: The Box Widget consists of twelve cylinders (A) connected in a hexahedral box (three-dimensional rectangle) with cylinders indicating on the edges of the box (B). In the middle of each face of the box is a sphere with a cylinder protruding from it (C) that provide resize control.

Purpose: The primary use of the Box Widget is to select a subvolume of the workspace for further manipulation (*e.g.,* volume rendering, isosurfaces, streamlines, mesh adaption) where the faces of the widget act as orthogonal clipping planes.

Controls: Clicking and dragging on one of the spheres rotates the widget about its center without changing the position of the center. Clicking on and dragging any resize handle causes the associated face to extend without changing its orientation. Dragging a cylinder causes the entire widget to move without changing its orientation.

5.4.4 Ring Widget

Appearance: The Ring Widget consists of a ring (A) with four embedded spheres (B), each with a resize cylindrical attached (D). Between two of the spheres is a sliding collar (C). One of the resize cylinders has a special material property (typically a different color from the other cylinders) to indicate that it is the "halfway point" for the slider (E).

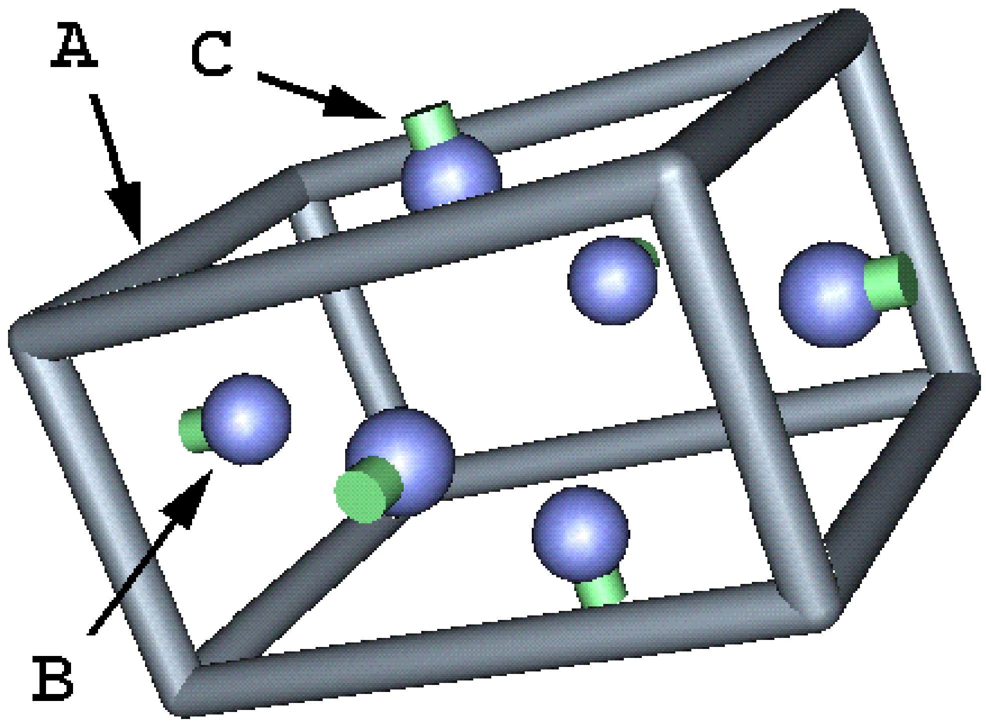


Figure 9: The boxwidget for selecting sub-volumes

Purpose: The primary use of the Ring Widget is to set the density of streamlines emerging from the ring—the ring serves as a set of seed points from which will emerge streamlines. The Ring Widget can also serve as a three-dimensional angle gauge, as a source for multiple streamlines throughout its surface, as a source for a stream surface from the outer ring, and as a source for a stream volume. Another use is as a color sheet, or "tie dye", in which the surface is colored as a function of the scalar value of the field at each point.

Controls: Clicking and dragging the slider collar along the ring changes the density of the seed points or some other related parameter. Dragging the spheres controls the orientation of the Ring Widget, while moving the resize cylinders changes the radius of the Ring Widget about its center. Dragging any other point on the ring moves the ring in space without changing its radius or orientation.

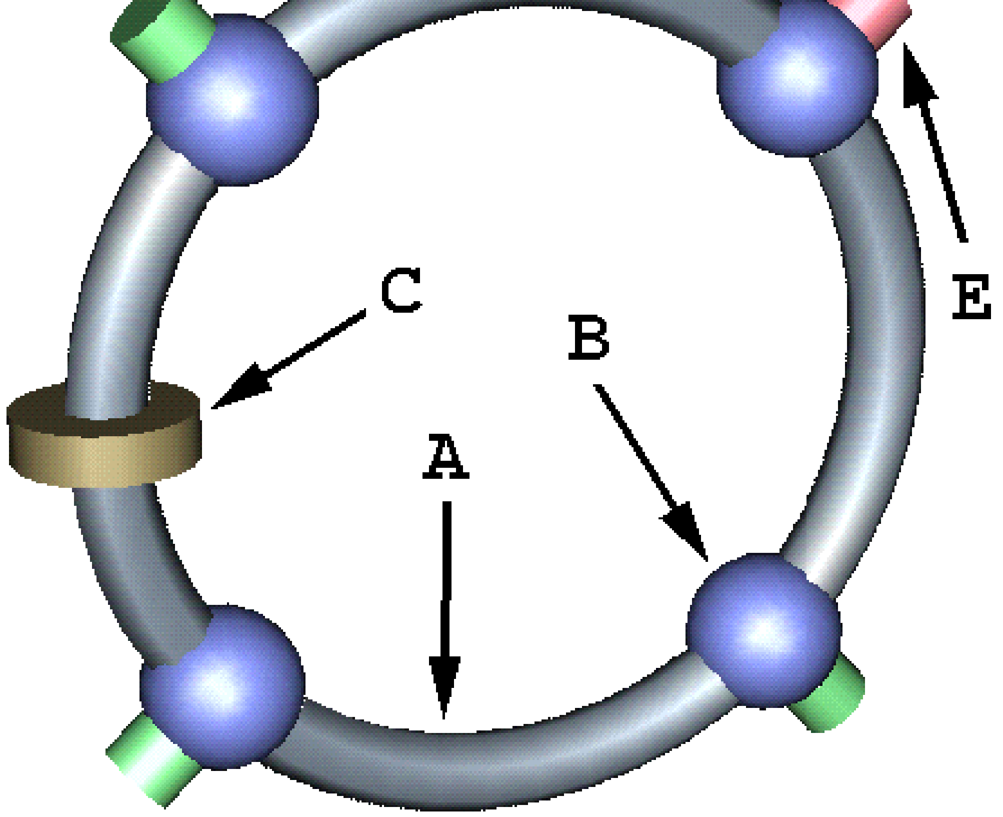


Figure 10: The ring widget for selecting cutting/projection planes

References

[1] J.A. Schmidt, C.R. Johnson, and R.S. MacLeod. An interactive computer model for defibrillation device design. In *International Congress on Electrocardiology*, pages 160–161. ICE, 1995.

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

[3] E.C. Anderson and J. Dongarra. Performance of LAPACK: A portable library of numerical linear algebra routines. *Proceedings of the IEEE*, 81(8), 1993.

[4] D.F. Allen, D.R. Haynor, and G.H. Bardy. 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*, pages 774–775. IEEE, 1991.

[5] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering scientific computations. *IEEE Computational Science and Engineering*, 1(4):44–62, 1994.

[6] J. Vetter and K. Schwan. Models for computational steering. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, 1996.

[7] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. *SIAM*, Aug. 1996.

[8] J.A. Kohl, P.M. Papadopoulos, and G.A. Geist. Cumulvs: Collaborative infrastructure for developing distributed simulations. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. Minneapolis, Mar. 1997.

[9] J. Vetter and K. Schwan. Progress: A toolkit for interactive program steering. In *Proceedings of the 24th Annual Conference of International Conference on Parallel Processing*, 1995.

[10] J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proceedings of the 11th International Parallel Processing Symposium*. Geneva, Switzerland, Apr. 1997.

[11] C.R. Johnson and S.G. Parker. A computational steering model for problems in medicine. In *Supercomputing 94*, pages 540–549. IEEE Press, 1994.

[12] S.G. Parker and C.R. Johnson. Scirun: a scientific programming environment for computational steering. In *Proc ACM IEEE Supercomputing Conf*, volume 2, pages 1419–1439. IEEE, Los Alamitos, CA, 1995.

[13] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[14] S.G. Parker, D.M. Beazley, and C.R. Johnson. Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59, 1997.

[15] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–44. Birkhauser Press, 1997.

[16] S. G. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999.

[17] J. Davison de St. Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: a massively parallel problem solving environment. In *IEEE Int Symp High Perform Distrib Comput Proc*, pages 33–41. IEEE, Piscataway, NJ, 2000.

Index