SCIRun Developer Guide

SCIRun 4.3 Documentation

Center for Integrative Biomedical Computing
Scientific Computing & Imaging Institute
University of Utah

SCIRun software download:

http://software.sci.utah.edu

Center for Integrative Biomedical Computing:

http://www.sci.utah.edu/cibc

Supported by:

NIH grant P41- RR12553-07

Author(s):
SCIRun Development Team

Contents

Chapter 1

SCIRun Overview

SCIRun is a computational workbench used for modeling, simulating and visualizing scientific problems, which is implemented in C++ with a Tcl/Tk GUI (Graphical User Interface). SCIRun uses a dataflow computational model. SCIRun provides algorithms, math and visualization tools implemented as discrete software units referred to as *modules*.

Modules are organized into packages: the default package, *SCIRun*, contains general-purpose tools. SCIRun can be used on Linux, Mac OS X and Windows platforms.

Chapter 2

Building SCIRun from Source

2.1 Prerequisites

- Subversion
- CMake (can create XCode projects, GNU make, NMake, Visual Studio)
- C/C++ compiler
- GNU make on Linux/Unix platforms

The free IDE Visual Studio Express is available for Windows.

2.2 Download Sources

2.2.1 Source Archive

Download the source archive from SCI software portal from the SCIRun software page. These are updated regularly on a weekly basis, and when a major bug fix has been applied. Only gzipped tar archives are available at the moment, so Windows users may need to download thirdparty tools to unpack the archive, such as Cygwin. The gzip product page also has a list of tools that can unpack the archive on Windows.

2.2.2 Subversion

Build SCIRun from the latest sources by checking out code from the SCI Subversion repository. We recommend reading Version Control with Subversion for those who have never worked with Subversion before.

Usually, the procedure is to check out the code from the repository trunk URL.

```
svn checkout --username anonymous \
  https://code.sci.utah.edu/svn/cibc/cibc/trunk SCIRun
```

Chapter 3

Modules

3.1 Tcl Interface

3.2 Communication between Tcl/Tk and C++

3.2.1 GUI Interface

SCIRun uses Tcl/Tk as its GUI front end. However, Tcl/Tk was not designed with a clean interface between Tcl code and C/C++. The purpose of the TCLInterface class in *Dataflow/GuiInterface* is to provide an abstraction layer make the task of moving data between the Tcl and the C++ portions of SCIRun transparent to the user.

Most of the code in the *Dataflow/GuiInterface* directory is used internally in SCIRun. The only exception are the GuiVars which can be access from both the Tcl and the C++ codes. These variables provide a transparent mechanism in C++ to set or get the Tcl variables they represent. Each such Tcl variable is associated with a Module and thus contains information about the module Tcl ID and a pointer to the module.

3.2.2 Programming with the SCIRun GuiInterface

On the Tcl side, the code should access the variables as regular Tcl variables. On the C++ side, the code needs to declare these variables inside a Module and access them via the `get()` and `set()` functions.

GuiVar

GuiVars are variables that encapsulate the interaction between the C++ code and the GUI code. The variable does not hold the actual value, rather it holds information which is used to access the corresponding variable on the GUI side. From the C++ side the user may set the variable value using the set() function and retrieve the value using the get() function. There are several specialization of the GuiVar class for particular variable types such as `GuiInt`, `GuiString` and `GuiPoint`.

In a module's GUI Tcl code:

```
itcl_class foo {
      ...
   method set_defaults {} {
     global $this-min
     global $this-max

     set $this-min 0                  # set to 0
     set $this-max [set $this-min]    # '[set $this-var]' returns its value
     }
 ...
}
```

In the C++ side, i.e. in the module class:

```
class Demo : public Module {
   ...
  GuiInt gui_min, gui_max;          // define GUI variables

  Demo( const clString& id );
   void init();
};

Demo::Demo( const clString &id )
: Module(...),
   gui_min("min",id, this),        // initialize a variable with
   gui_max("max",id,this)          // its name on the Tcl side.
 {
   ...
 }

 void Demo::init()
 {
   gui_min.set(7);                // set a Tcl variable
   int i = gui_max.get();         // get a value from the Tcl side
 }
```

Chapter 4

Datatypes

There are four basic datatypes defined in SCIRun: Field, Matrix, ColorMap, String. The Nrrd datatype is provided by the Teem package, which wraps the Teem thirdparty project. The DICOM datatype inteface is provided by Kitware's Insight Toolkit (ITK).

Chapter 5

Import and Exporting File Formats

SCIRun can read in and write out a variety of file formats through a plugin framework.

Chapter 6

SCI Coding Standards

6.1 Required Coding Standards

- All code and comments are to be written in English.
- All files must include appropriate MIT license information, which must appear at the top of the file. Please copy from the LICENSE file at the top of the source tree. Please update the year to keep the license current.
- Include files in C++ always have the file name extension **.h**. Use uppercase and lowercase letters in the same way as in the source code.
- Implementation files in C++ always have the file name extension **.cc**.
- Every include file must contain a *guard* that prevents multiple inclusions of the file, for example:

```
#ifndef CORE_GEOMETRY_BBOX_H
#define CORE_GEOMETRY_BBOX_H 1

// Code...

#endif
```

- The name of the guard should be of the following form: `DIR_DIR_FILENAME_H`
- Use forward declarations wherever possible as opposed to including full definitions for classes, functions, and data:

```
// Class
class PointToMe;

// Function
```

```
void my_function(PointToMe &p, PointToMe *ptm);

// Data
PointToMe *m;
```

Never include /usr/include/*.h, for example iostream.h in any header file. This causes a huge amount of code to be recursively included an needlessly compiled. Use forward declarations to avoid this.

- The names of variables and functions will begin with a lowercase letter and include underscore word separators. Names of constants should be in all CAPITALS, with underscore word separation:

```
static int CONSTANT_INT_FIVE = 5;
void my_function_name();
int my_local_variable_name = 0;
```

- The names of class member variables are to end with an underscore (_):

```
class MyClass {
  int myClassMember_;
};
```

- The names of abstract data types (that is, classes), and structs are to begin with an uppercase letter, and each new word in the name should also be capitalized.

```
class MyNewClassName {
  // ...
};
```

- All member functions which do not change an object's state should be declared `const`
- Constants are to be defined using `const` or `enum`. Never use `#define` to create constants.
- A class which uses `new` to allocate instances managed by the class **must** define a copy constructor and an assignment operator.
- Classes should never assume that the input is perfect and a sensible number of safety checks should be in place to detect faulty inputs.
- Use exception handling to trap errors (although exceptions should only be used for trapping truly exceptional events).

- Our exception model includes two levels of exceptions. The top level of exceptions are defined in *Core/Exceptions/Exceptions.h* and are thrown when a class specific exception is not appropriate. The bottom level of exceptions are class specific, defined in the class that throws them, and are subclassed off of the top level exceptions. These class specific exceptions are exceptions that can be caught and handled from the calling function (1 level above the class). However, if the calling function chooses not to (or cannot) handle the class specific exception, the exception will propagate to the next level at which point it can be trapped and handled in the form of a top level exception. An example of a class specific exception would be a StackUnderFlowException for a stack class.
- Do not use identifiers that begin with an underscore, such as `_myBadIdentifier`.
- Do not use `#define` to obtain more efficient code; use inline functions instead.
- Avoid the use of numeric values (magic numbers) in code; use symbolic values instead. This applies to numeric values that are repeated within the code but represent the same value, for example `MAX_ARRAY_SIZE = 1024`.
- Do not compare a pointer to `NULL` or assign `NULL` to a pointer; use 0 instead as `NULL` is not part of the C++ standard and is not guaranteed to be defined.
- Avoid explicit type conversions (casts). However when a cast is needed, an explicit cast is preferred over having the compiler decide which kind of cast to do. Use C++ casts (`static_cast`, `dynamic_cast` etc.) rather than C-style casts.
- Never convert a constant to a non-constant. Use `mutable` if necessary, but be aware of the thread safety problems this causes.
- Never use `goto`.
- Do not use `malloc`, `realloc`, or `free`. Use `new` and `delete` instead. Allocate arrays on the heap using `new []` and `delete []`:

```
int *myArray = 0;
myArray = new int[256];
...
delete [] myArray;
```

- Do not use `long`. Use `int` for 32-bit integers and `long long` for 64-bit integers.
- Use C++ STL classes whenever possible instead of writing novel containers.
- Do not use C style converters such as `atoi` because they are not consistently threadsafe on every platform SCIRUN supports. Use the converters in SCIRun's StringUtils file or STL converters.

- Be aware that longs, floats, doubles, long doubles etc. may begin at arbitrary addresses. Do not assume that built-in data types are contiguous in memory.
- Always use plain `char` if 8-bit ASCII is used. Otherwise, use `signed char` or `unsigned char`.
- Do not assume that a char is signed or unsigned.
- Do not depend on underflow or overflow functioning in any special way.

6.2 Recommended Coding Standards

- Avoid using more than 80 columns per line.
- Do not format using tabs. Set IDE and text editor preferences to use spaces instead of tabs.
- Group local includes together, then group system includes together.
- Avoid global data if possible.
- Optimize code only if you know that you have a performance problem. Think twice before you begin.
- When developing new code, always force your compiler to compile with the maximum warning setting, and before you check in code, fix all warnings.
- Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another. For example, see *Core/Thread*.
- Encapsulate global variables and constants, enumerated types, and typedefs in a class.
- Functions in general should not be more than 30 lines long (excluding documentation and indentation). If you find this situation, break the function into several smaller functions.
- If a function stores a pointer to an object which is accessed via an argument, let the argument have the type pointer. Use reference arguments in other cases.
- When overloading functions, all variations should have the same semantics (be used for the same purpose).
- Do not assume that you know how the invocation mechanism for a function is implemented.
- Do not assume that an object is initialized in any special order in constructors.

- Use a `typedef` to simplify program syntax when declaring function pointers or templated types.
- When two operators are opposites (such as == and !=), it is appropriate to define both.
- Pass function arguments by reference or by constant references (`const &`) instead of by value, unless using a built-in data type or a pointer.
- Minimize the number of temporary objects that are created as return values from functions or as arguments to functions. Do not write code which is dependent on the lifetime of a temporary object.
- Use C++ streams (`std::cout`, `std::cerr`) instead of printf.
- Avoid the use of `using namespace std;` (or other namespaces) in include files, as they can spill into other files with unintended consequences.
- Try to use smart pointers (`Handles`) to automatically deallocate memory when an object is not needed anymore.
- When a function is pure, i.e. it does not modify any of the class members, annotate it as such so it can be used safely in multi-threaded code.
- When variables are being shared between threads always use a `Mutex` for access control. Use a `Guard` whenever possible to ensure that mutexes are unlocked when going out of scope.
- Use the STL string class and not C-style strings whenever possible.

6.3 Memory Management

6.3.1 Avoiding Memory Leaks

Use the SCIRun reference counted LockingHandles wherever possible, which ensures that memory will be freed when the handle goes out of scope and the reference count is 0. The LockingHandle class is defined in *Core/Containers/LockingHandle.h*. For instance, SCIRun datatypes, if passed as data through a dataflow port are wrapped in a LockingHandle.

Chapter 7

Further Reading

7.1 Useful C++ References

- The C++ Programming Language by Bjarne Stroustrup.
- Thinking in C++ by Bruce Eckel
- Effective C++ by Scott Meyers