VE-Suite Code Guidelines

Basic quidelines are needed so that your code looks properly formatted in someone else's editor and is easy to follow. We modelled these guidelines after those of two open-source projects that we use heavily, VTK and VRJuggler .

Class Structure

We only put one public class per file.

Every class name, macro, etc. starts with either "`cfd`" or "`CFD`" to avoid name clashes with other libraries. Classes should all start with "`cfd`" and macros or constants can start with either.

Class names and file names are the same (e.g., cfdContours class is declared in cfdContours.h and implemented in cfdContours.cpp). This makes it easier to find the correct file for a specific class.

We try to keep all instance variables protected. The user and application developer should access instance variables through Set/Get methods.

We **always** initialize member variables in the constructor. Too many problems are eventually traced back to a pointer or variable left uninitialized.

Classes should implement a destructor, copy constructor, and equals operator.

Names

Make names clear. Try to spell out a name and not use abbreviations. This leads to longer names but it makes using the software easier because you know that the `SetRasterFontRange` method will always be called that, not `SetRFRange` or `SetRFontRange` or `SetRFR`. When the name includes a natural abbreviation such as OpenGL, we keep the abbreviation and capitalize the abbreviated letters.

We only use alphanumeric characters in names, [a-zA-z0-9]. We do not use underscores ('_'), so names like `exterior _ Surface` or `_ exteriorSurface` are not welcome.

We use capitalization to indicate words within a name. For example, a class could be called `cfd-VectorTopologyFilter`.

Use "`this`" inside of methods even though C++ doesn't require you to. This really seems to make the code more readable because it disambiguates between instance variables and local or global variables. It also disambiguates between member functions and other functions.

Class Names

Classes names are nouns and usually start with "cfd".

Variable Names

Variable names are nouns and should begin with a lower case character.

Function Names

Function names are verbs (e.g., `SetRasterFontRange`) and start with an upper case letter.

Functions and Function Arguments

Use const on any member function that does not change any data. It enforces the purpose of a function and allows for better compiler optimizations.

C++ style variable usage

Declare variables where they're first used. C++ lifted this restriction from C for a reason. It makes code more modular and easier to understand, modify, and debug.

Braces

Place brace under and inline with keywords:

```
if ( condition )      while ( condition )

{                     {

   ...                   ...

}                     }
```

Justification

If you use an editor that supports brace matching (such as vi or vim), then it is easy to hop around long blocks.

If you want to comment out the conditional clause, it only takes one "//"

Use braces for all if, while and do statements even if there is only a single statement within the braces. It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. It doesn't affect execution speed. It's easy to do. It will keep you out of trouble when using vrjDebug print statements.

Header Files

Header files are for declarations only. It is difficult to read code and find code if source is in the declarations.

The header file of the class should include only the superclass header file. If you need any other includes, include comment at each one describing why it should be included.

Use forward class declarations to eliminate the other includes that you are tempted to insert.

Header files should use guards to prevent multiple inclusion. These guards should be defined in the style `FILENAME _ H`, which is the filename written in all upper case with punctuation such as dots ('.') replaced by underscores ('_')

```
#ifndef CFDHEADER _ H

  #define CFDHEADER _ H

  ...
```

```
#endif // CFDHEADER_H
```

Comment the end of every #endif, as shown above. Nested #ifdef's can be difficult to follow.

A new line after the last endif if is required by some compilers.

Text Formatting

Limit lines to 80 characters. That allows you to open and see multiple files on a single display and you can better compare two files in xdiff or tkdiff. Set your editor to default to an 80-character wide screen.

Three (3) space indentation – Set your editor to insert three spaces every time you hit the tab key. True tabs ("\t") are pretty much just for Makefiles.

Insert whitespace for readability. This means that one blank line should occur between function definitions or between different blocks of code.

Namespaces

Don't place the "`using namespace`" directive at global scope in a header file. This can cause lots of invisible conflicts that are hard to track. Keep "`using`" statements to implementation files (if at all).

For standard namespace keywords (cout, cin, cerr, endl, vector, string), You must do "`std::cout`", etc.

Comments

Comments must tell "why", not "what is happening". At every point where you had a choice of what to do, place a comment describing which choice you made and why.

Parens () with Key Words and Functions Policy

Do not put parens next to keywords. Put a space between keywords and parens.

Do not use parens in return statements when it's not necessary.

Do put parens next to function names.

Justification: Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

Example:

```
if ( condition )      while ( condition )

{                     {

   ...                   ...

}                     }

strcpy( s, s1 );      return 1;
```

Array Indexing

Single-character variable names can easily be mistaken for numbers (“i”s can sometimes look a lot like ones). Make the array index stand out with whitespace:

```
array[ i ] = 0.0;

array[ 1 ] = 0.0;

array[ 2 ] = 0.0;
```

Submit clean code

Make sure your code compiles without any warnings with -Wall and -O2