SimChemistry for Windows: Design Overview

Author:	Charlie Wartnaby (chem @simchemistry.co.uk / charlie @wartnaby.org)
Version:	3
Date:	09 Apr 2016
Copyright:	© Charles Wartnaby 2016

Contents

1. Introduction 3
 1.1 References 3
 1.2 Terms and Abbreviations 3
2. Programming Environment 3
 2.1 Compiler/Development Environment 3
  2.1.1 Current Required Tools 3
 2.2 Language 4
 2.3 Microsoft Foundation Class Libraries 4
 2.4 Random Number Generator 4
 2.5 Installation Package 4
 2.6 File Structure 5
 2.7 Naming Conventions 5
 2.8 Debug Support 5
 2.9 Help file 6
3. SCW Windows Application Architecture 6
 3.1 Application 6
 3.2 Document 6
 3.3 Views 6
 3.4 Class Summary 7
4. Simulation Objects 8
 4.1 Class Summary 8
 4.2 Object Lists and Structures 9
 4.3 Object Types 9
 4.4 Interaction Types 9
5. The Simulation Engine 9
 5.1 Calling and Iteration 10
 5.2 Simulation Phases on Each Single Step 10
  5.2.1 Preparation Step 10
  5.2.2 Maintenance of Neighbour Lists 10
  5.2.3 Calculation of Forces 10
  5.2.4 Accelerations 11
  5.2.5 Prediction of Discrete Events 11
  5.2.6 Execution of Discrete Events 11
  5.2.7 Synchronization 11
  5.2.8 Update of Monitors and Graphs 11
 5.3 Display 11

1. Introduction

This document explains the structure and programming environment for v2.2 of SimChemisty for Windows, to give developers sufficient understanding to sensibly make alterations to the package.

It does not give function-by-function detailed design information. Detailed design is presented only as source code comments, which are extensive in the case of mathematical derivations. This document provides only an overview and does not attempt to duplicate what is already explained in the code files.

Much of this document considers issues of software structure and environment. How SCW actually works as a numerical simulation is described in section 5 below.

SCW can be downloaded from http://www.simchemistry.co.uk. If that no longer exists, try http://www.wartnaby.org/.

1.1 References

[Ref 1] "The Art of Molecular Dynamics Simulation", D.C. Rapaport (1995, Cambridge University Press)

[Ref 2] "Numerical Recipes in C; the Art of Scientific Computing", W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P.Flannery (2nd edithion; 1996, Cambridge University Press)

[Ref 3] "The C++ Programming Language", Bjarne Stroustrop. (3rd edition;1997, Addison-Wesley)

1.2 Terms and Abbreviations

MDI	Multiple Document Interface (a type of Windows application)
MFC	Microsoft Foundation Classes (a Windows programming environment)
SCW	SimChemistry for Windows
SDI	Single Document Interface (a type of Windows application)
VB	Visual Basic

2. Programming Environment

2.1 Compiler/Development Environment

SCW previously built under Microsoft Developer Studio Visual C++ v6.0. Developer Studio had components for various languages (Visual Basic, Java) and tools (Visual SourceSafe), but only the C++ package was required here.

Historically, SCW started in MSVC1.5 and then MSVC5.0.

The original development used the Installshield system (bundled with MSVC6) to build an installer. No installer builder is currently set up to work with the latest tools.

2.1.1 Current Required Tools

SCW currently builds and executes using the free Visual Studio 2015 Community Edition, installed using the MFC application support option. (A number of warnings are now emitted due to the use of deprecated functions etc, but nevertheless it builds.)

The Multimedia library included in MSVC6 was used to give accurate timing. The current version of winmm.lib now included was taken from "C:\Program Files (x86)\Microsoft SDKs\Windows\v7.1A\Lib" on a Windows 10 PC.

Microsoft Help Workshop https://www.microsoft.com/en-gb/download/details.aspx?id=21138 is used to convert original WinHelp documentation (.rtf and .bmp fles) to a .chm project. WinHelp format, in which the SCW help was written, is no longer supported in recent versions of Windows.

2.2 Language

SCW was written in C++ for the following reasons:

- the problem of getting a range of different types of simulation object to perform actions, display themselves, and so-on in a structured way lent itself naturally to a class-based language;
- C++ provided essentially the same performance as C, whereas Visual Basic would have been very slow given the tools and PC hardware available at the time;
- the easiest Windows application structure at the time (MFC) was C++ based anyway.

The C++ is quite "early" in style, dating back to around 1996. This predates Stroustrop's 3rd edition [Ref 3], so modern features such as iterators, exceptions and different types of cast are absent. However, this should make it more easily accessible to C programmers.

The key principle to understand for C++ novices is *inheritance* between *classes*. A class is like a struct in C, essentially containing data and pointers to functions. An *instance* of a class (or *object*) is one specific real example of it within the program (e.g. one molecule in SCW).

A *base class* might provide some generic functions and specify that certain functions should exist, even if it does not provide them itself. A *child class* can then be *derived* from the base class that still retains (*inherits*) all of the functions and data provided by the base class, but which may also add its own functions and data, and which will *override* functions defined by the base class with specific versions suitable for its own needs. Crucially, given a list of objects all derived from a base class, we can ask each of them to perform some function (e.g. 'draw yourself') and the correct function for each is called, be it for a wall or a molecule.

2.3 Microsoft Foundation Class Libraries

Microsoft have invented various methods for C or C++ programs to interact with the Windows operating system over the years:

SDK (software development kit): C language API; used in some places in SCW (examples??)

- MFC (Microsoft Foundation Classes): library of C++ classes which "wrap" SDK functions and provide a structure in terms of an application with documents and document views.
- ATL (Active Template Library): C++ template wrappers around Windows SDK, with more emphasis on Component Object Model (COM)
- .NET: most recent offering, giving a single common framework that can be used with different languages (C++. C#, VB, ...)

SCW uses the MFC application structure. Within MSVC6, extensive help documentation and tutorials (especially the Scribble application) exist to explain how the architecture works and what the many classes offer.

2.4 Random Number Generator

SCW 2.2 and previous versions used the ran1 routine from Numerical Recipes in C, with permission. However, some other routine must be substituted now because distribution of the source code is not permitted by the publishers of Numerical Recipes. Only a simple flat distribution in the interval 0 to 1 is required. Currently the system library is used instead.

2.5 Installation Package

The free edition of InstallShield, which comes bundled with MSVC6, is used to make the executable installation package (setup.exe and associated files). The installation process is configured by a script file which is responsible for guiding the user through the installation steps and registering the application.

2.6 File Structure

The Windows application is in the 'code' subdirectory, which is the default directory when the MSVC workspace is opened.

The easiest way of finding your way around is to grep ("Find in Files") for identifiers.

In particular, the body of C++ functions can be most easily found by searching for the scope resolution operator (two colons, ::) before the function name. For example, search for "::OnIdle" to find where that function is implemented, or ::CalcForces() to find the corresponding function for each type of object.

In general, each class has a header (.h) and implementation (.cpp) file. The header file provides the interface to that class for use by other code, and the implementation file contains its executable code. However, in some cases small classes are lumped together in larger files.

SCW started in the era of "8.3" MS-DOS names, but as 32-bit Windows appeared more descriptive long filenames came into use. A mixture of filename styles persists, which avoided the effort of getting it to build after a file name changing exercise.

2.7 Naming Conventions

Following MSVC convention, classes all begin with 'C', e.g. CSim, CPiston.

Member variables of classes all begin with 'm_', e.g. **m**_dDeltaTime.

Data types are generally indicated by a prefix:

Prefix	Type	Example
b	BOOL	m_**b**InClientRect
n	int or enum	m_**n**Type
l	long	m_**l**TotalNumObjs
d	double	**d**DeltaTimeOverMass
s	CString	m_**s**Name
p	pointer	**p**Mol
fs	flag set	m_**fs**Valid
num	CNumeric	m_**num**Radius
pt	CPoint	m_**pt**LogDragStart

However, though locals of limited and obvious scope do not always use that convention.

2.8 Debug Support

Run-time checking is used extensively using the ASSERT() macro. In debug builds, this should help identify invalid conditions that might be inadvertently introduced by making changes.

Flag structures such as CMol::m_fsValid track which quantities are valid and which are not in some cases (e.g. the velocities and direction cosines of molecules). This avoids computing quantities which are not required by skipping square root and trigonometric operations, to improve simulation performance. ASSERT() is then used to check that these quantities are valid before they are used elsewhere.

The TRACE() macro is also used to report program progress in debug builds.

In various places, global_debug_counter is incremented. This is to help trace difficult bugs concerning simulation behaviour. For example, if a molecule is seen to 'leak' through a wall in some saved simulation, a debug build can be run and stopped after the errant behaviour has been seen. The value of global_debug_counter can be checked in the debugger. The program can be restarted, this time with the debugger set to halt execution at a somewhat lower value of global_debug_counter. After a little experiementation, the value at which the problem shows itself can be found. Temporary code can be added to break on this value to make it easier to check data structures, calling sequences and values where the problem occurs.

All of these disappear in release builds, to improve performance and reduce code size.

2.9 Help file

The original help is in WinHelp (*.hlp) format, which is now obsolete. This is constructed using rich text format (RTF) files.

The file afxcore.rtf needs to be edited to modify the online help. The help build tools in MSVC6 seem quite particular; editing the .rtf file in Word 97 seems OK, but using write.exe is not.

Using modern tools to build the help instead in .chm format, in Microsoft HTML Workshop do this:

- File... New Project
- Tick "Convert WinHelp Project"
- Browse to code\lennard.hpj for the WinHelp project file, and code\lennard.hhp for where you want the HTML help project to be created
- Do File... Compile to rebuild Lennard.chm.

The program has now been updated to launch the .chm file if any help links are used. Unfortunately this means it no longer goes directly to the relevant section in the help.

3. SCW Windows Application Architecture

This section explains how the program fits together as a Windows application, with little on the actual simulation.

3.1 Application

SCW is a Windows application using the multiple document interface (MDI) structure provided by the Microsoft Foundation Classes (MFC) environment. This means that more than one document can be open simultaneously, in different child windows, within the overall application window (similar to MS Excel.)

The MFC environment also allows a simpler single document interface (SDI) structure, of which Notepad is an example.

The application is an instance of the CLenApp class, of which there is only one. The "main frame" window that the application presents on screen is an instance of the CMainFrame class, which owns the menus and toolbars. These user interface elements are mostly targeted at whichever document is currently active, but some are handled by the application itself (e.g. File... Exit).

3.2 Document

Each document (CLenDoc class, derived from CDocument) contains exactly one simulation (CSim class). A document and a simulation are almost interchangeable. However, all the simulation structure is in the CSim instance.

3.3 Views

Each document has two possible views: the graphical simulation view and the console view. These are two ways of looking at the same document (like Draft or Page Layout view in MS Word).

The simulation view uses a "double buffered" method to produce smooth graphics. The new positions of all the simulation objects are drawn in turn onto a blank bitmap in memory, and only when it is completed is it copied to the screen.

The console view is more complex in that it is a 'split' window, composed itself of two child windows (one to type commands into, the other to display the script listing).

Within Windows, the DirectX library would now provide a higher-performance structure for graphical display, including 3-dimensional elements.

3.4 Class Summary

Class	Derived From	Files	Notes
CLenApp	CWinApp	lennard.h/cpp	The Windows application class; there is one instance of this class when the application runs, defined statically in lennard.cpp.
CMainFrame	CMDIFrameWnd	mainfrm.*	The main window which bounds the application on the screen, and owns the menus and toolbars. Handles some commands (e.g. File... Exit).
CLenDoc	CDocument	lendoc.*	One of these per open document (i.e. simulation). This handles user interface commands specific to the current simulation, and is responsible for saving or opening a simulation from disk.
CSimView	CView	simvw.*	Graphical view which presents the current document (i.e. simulation) in a child window of the application.
CConsFrame	CMDIChildWnd	consfrm.*	Window containing the split console view (place to type in commands below, and view of commands already in script above).
CConsoleView	CScrollView	consvw.*	Displays the list of commands already in the script in the console view.
CCmdView	CEditView	cmdvw.*	Provides the edit box in the lower part of the console view where the user may enter new commands using the keyboard.

4. Simulation Objects

4.1 Class Summary

Class	Derived From	Files	Notes
CSim	CSimObj	sim.*	The complete simulation in the current document. Contains the simulation "engine".
CSimObj	(none)	simobjs.*	The base class for all simulation objects (both object types, and specific object instances). Includes functions for parsing a command targeted at that object and saving itself to disk. Everything is saved in the form of text commands, just as if typed by the user or read from a script.
CObjType	CSimObj	simobjs.*	The base class for each type of simulation object (e.g. a *kind* of wall or molecule). One instance created for each type of object present in the simulation.
CSObj	CSimObj	simobjs.*	The base class for each actual simulation object (e.g. a specific wall or molecule). One instance created for each individual object present in the simulation.
CInt	CSimObj	baseint.*	Base class for each *interaction* between any two *types* of simulation object. Interactions can be hard (billiard-ball) or soft (with forces).
CNP	(none)	simobjs.*	A neighbour pair object, tracking the interaction between two specific objects in the simulation.
CListHandle	(none)	simobjs.*	A 'handle' belonging to a CNP object so that each CNP can be included in two different linked lists (e.g. for a molecule/piston interaction, one handle is to include in the list of molecules and the other in the list of pistons).
CNumeric	(none)	simobjs.*	One instance for each numerical quantity present in a simulation that can be displayed in a text box or plotted in a graph, etc. Can be contained within the simulation, object type or specific objects. (E.g. the simulation timestep, temperature of a wall type, or position of a piston.)
CNumLink	(none)	simobjs.*	Represents a section of textbox string, optionally linking to a live CNumeric quantity.
CHotSpot	(none)	simobjs.*	Some objects have defined rectangles which the simulation

Class	Derived From	Files	Notes
			should change cursor for if it is placed over them, and do things when they are clicked

4.2 Object Lists and Structures

The efficient execution of the simulation (see 0 below) depends on maintaining lists and trees to allow the immediate location of simulation objects in different circumstances.

For hard interactions, a tree of neighbour pair (CNP) objects is maintained tracking which collisions will happen in what order so that they can be processed in turn.

4.3 Object Types

Type Class	Specific Instances	Files	Notes
CObjTypeMol	CSObjMol	mol.*	A molecule.
CObjTypeWall	CSObjWall	wall.*	A containing wall.
CObjTypePiston	CSObjPiston	piston.*	A moving piston which collides with molecules, applying pressure.
CObjTypeMonitor	CSObjMonitor	monitor.*	A measurement/control region providing statistics on the molecules it contains, and optionally controlling properties.
CObjTypeGraph	CSObjGraph	graph.cpp, graphx.h	A live graph displaying numeric data.
CObjTypeSlider	CSObjSlider	slider.*	A slider control to allow the user to vary input quantities.
CObjTypeTextbox	CSObjTextbox	textbox.*	A text label which may include live-updating numeric quantities.

4.4 Interaction Types

Type Class	Files	Notes
CNullInt	nullint.*	A do-nothing interaction (for when there is no interaction between two types of object).
CLennardJonesInt	mmljint.*	Lennard-Jones potential "soft" interaction between two types of molecule, providing accelerating forces.
CMolMolHardInt	mmhrdint.*	A hard collision interaction between molecules with elastic scattering following classical Newtonian mechanics.
CMolPistonHardInt	mphrdint.*	A hard collision interaction between pistons and molecules.
CWallMolDefaultInt	wmdefint.*	Default wall-molecule interaction, again with hard Newtonian collisions.
CMonitorMolDefaultInt	mnmldint.*	Molecule/monitor interaction which handles the monitor's ability to make statistical measurements of the molecules it contains, or to control them.

5. The Simulation Engine

SCW combines discrete event simulation (hard collisions) and classical molecular dynamics (calculation of forces with integration of accelerations), allowing both types of interaction to be in use simultaneously in one simulation.

The Art of Molecular Dynamics Simulation [Ref 1] gives very useful guidance on algorithms for both discrete event and force-based simulation, and SCW draws upon the ideas (but not the code) presented in that text. The code references this book in places.

5.1 Calling and Iteration

The simulation is called in the "idle loop" of the application, which is intended to handle background tasks such as displaying menus and toolbars in their current state. (See CLenApp::OnIdle() and CSim::OnIdle().)

The Windows documentation specifically recommends *not* executing intensive tasks in this manner! However, it provides a convenient means of having the simulation execute as fast as possible, while giving priority to user interface actions, without the complexities of a multi-threaded architecture.

On each call to CSim::OnIdle(), a number of simulation steps may be performed before one update of the graphics. This is also responsible for stepping through the stored script and allowing progress to be paused, resumed and delayed.

The simulation timestep to use can be set manually by the user. Otherwise, a fairly long default value is used for simulations containing only "hard" interactions, where the integration of accelerations is not an issue. For simulations containing "soft" interactions, an empirically-derived algorithm is applied to choose a timestep that is short enough to maintain realistic behaviour (conservation of energy) but long enough to keep the simulation speed reasonable. See CSim::ChooseDeltaTime().

5.2 Simulation Phases on Each Single Step

One actual molecular dynamics timestep is performed on each call to CSim::SingleStep().

5.2.1 Preparation Step

PrepStep() is called for each type of simulation object.

Each simulation object is considered in turn. Its acceleration is zeroed and the highest speed of any object is determined. This maximum speed is used to decide when the neighbour lists need updating.

5.2.2 Maintenance of Neighbour Lists

If all possible interactions between all possible objects were considered, then the simulation would be an N^2 problem, quickly becoming very slow for large simulations.

To avoid this, SCW maintains a list of pairs of neighbouring objects that *may* interact in the near future, and considers only the interactions between those. Given the highest object speed currently found of any object, the interactions between objects which are too far apart to interact in the near future are excluded. The cutoff distance for any Lennard-Jones interactions that have been defined is taken account of, so setting a large cutoff distance will considerably slow a large simulation.

The list of neighbour pairs has to be recomputed before any object could have feasibly moved far enough to become a neighbour of an object that was previously too far away to consider. Rebuilding the list is an N^2 problem but is not too demanding computationally.

A "safety distance" is defined, which is the largest distance between objects for which interactions will be considered. At each timestep, this is eroded by the distance the fastest object may have travelled *doubled* (as it may be approaching another object at the same speed). The neighbour pair lists are rebuilt when this is eroded below an "insurance distance".

A neighbour pair list is maintained by each interaction currently in use, and the work of computing the list is delegated to each interaction which "knows" the details of how to calculate the required separation of objects it understands. See BuildNeighList() function for each interaction.

5.2.3 Calculation of Forces

For each interaction that involves forces (i.e. Lennard-Jones interactions), each neighbour pair is considered in turn and the force between the affected objects computed. See CalcForces() for each interaction.

At the same time the virial contribution to pressure [Ref 1] is computed, and the potential energy for each molecule.

5.2.4 Accelerations

Given the forces acting on each object (through interactions, gravity, or piston springs) the acceleration is calculated and integrated to update the velocity accordingly. See ApplyAccels() for each object type.

5.2.5 Prediction of Discrete Events

For each "hard" interaction, all neighbour pairs are checked to see which objects are due to collide (or react, etc) in this timestep. This work is delegated to each interaction, which "knows" the details of how to compute events for the object types it is concerned with. It may record a "hint" for each predicted collision (e.g. which corner of a piston a molecule is about to hit).

For each collision predicted in this timestep, an entry is made in a pointer-linked tree structure owned by the simulation which orders the collisions in time as they are built up. The tree starts at CSim::m_pRootNode.

See also chapter 12 (Step Potentials) in [Ref 1].

5.2.6 Execution of Discrete Events

The predicted tree of discrete events is then processed, starting with the earliest in time. See CSim::ExecHardEvents().

The first event is executed by moving the affected objects to the point of the collision, and setting their velocities to the post-collision values. The impulsive contribution to pressure is accounted for at this point.

Every neighbour pair attached to either of the objects that has just collided then has all of its possible collision events recomputed, as in general any previously predicted events will now be invalid.

In the case of reactions, objects may be destroyed or created. To keep the data structures consistent without undue complexity, further processing of discrete events in the current timestep is halted if a reaction occurs. (See CSim::InvalidateAllHardEvents().)The events are all recomputed on the next timestep instead.

5.2.7 Synchronization

Following the execution of discrete events, different objects will be placed at positions correct for different times (the time of their last collision, or the time of the beginning of the timestep if they have nto been involved in discrete events).

All objects must therefore be moved according to their current velocities to bring them to their final positions for the end of the current timestep.

5.2.8 Update of Monitors and Graphs

With objects in their final positions, monitors update their measurements. The computed values are then available for display in graphs and textboxes.

5.3 Display

After a number of single simulation steps have been performed, the graphical display is updated by calling CDocument::UpdateAllViews().

This in turn causes the OnDraw() function to be called for each view in the current document.

The graphical representation of the simulation is redrawn by CSimView::OnDraw().