Introducing Design Patterns, Graphical User Interfaces and Threads within the Context of a High Performance Computing Application

James Roper and Alistair P. Rendell*

Department of Computer Science, Australian National University
Canberra ACT0200, Australia
alistair.rendell@anu.edu.au

Abstract. The cross fertilization of methods and techniques between different subject areas in the undergraduate curriculum is a challenge, especially at the more advanced levels. This paper describes an attempt to achieve this through a tutorial based around a traditional high performance computing application, namely molecular dynamics. The tutorial exposes students to elements of software design patterns, the construction of graphical user interfaces, and concurrent programming concepts. The tutorial targets senior undergraduate or early postgraduate students and is relevant to both those majoring in computing as well as other science disciplines.

1. Introduction

By its very nature computational science is interdisciplinary requiring mathematical, computing and application specific skills. At most tertiary institutions, however, the undergraduate curriculum funnels students towards specialization. Accepting this raises the obvious question of how existing courses can be modified to show students that skills learnt in one domain may be applied to another, or alternatively, how techniques developed in another area might be useful in their field of study.

The above divide is particularly noticeable between students majoring in some aspect of computing versus those majoring in other science subjects – like chemistry, physics or mathematics. Thus while the chemistry, physics or mathematics student may take a class or two in computer science during their freshman year, timetable constraints and pre-requisite requirements often inhibit them from taking higher level computer science classes. Likewise a student pursuing a computer science major may get some exposure to first year chemistry, physics or mathematics, but rarely do they progress to higher level courses. While some knowledge of a discipline at first year is useful, computer science freshman courses generally teach little more than basic programming, with more advanced concepts like software design and analysis or concurrent programming left to later years.

Noting the above, and as part of the computational science initiative of the Australian Partnership in Advanced Computing [1] we have been working to design a series of tutorials that can be used by senior students in both computer science and

other science courses. The aim is to construct a tutorial that, depending on the situation, can be used to expose computer science majors to aspects of scientific programming, or introduce aspects of computer science to students with other science majors. This paper outlines our work on one such tutorial, where the objective is to illustrate the use of design patterns, graphical user interfaces and threading in the setting of a traditional high performance computing application code. Molecular dynamics was chosen as the application domain since it is relatively easy to comprehend with little science or mathematics background.

2. Tutorial Background

The tutorial assumes that the reader is a competent programmer, but does not assume that they have experience in any particular programming paradigm, thus concepts like inheritance and threading are described. Visual Python (VPython) [2] was chosen as the programming language. This is a 3-D graphics system that is an extension of the Python programming language and has the advantage that the programmer need not worry about the underlying mechanism of how to build the display environment, but is just required to specify the layout of the shapes used in any particular model. (A subsequent tutorial, currently under development, will include mixed programming using VPython and C/C++.)

While the tutorial does assume a working knowledge of VPython, when this is lacking students are referred to two earlier tutorials [3] that were developed at the Australian National University and begin by modeling a “bouncing ball” confined within a box, and then evolve into a simulation of hard sphere gas particles. These earlier tutorials assume little or no programming experience and lead the user through the process of creating a ball (atom), setting it in motion, confining it to a box, and eventually to performing a gas simulation and comparing the simulated velocity distributed with the expected Maxwellian distribution. These earlier tutorials have frequently been given to students in grades 11 or 12 and their teachers with great success, demonstrating the ease of programming with Python and the interactivity imparted by using VPython to display results.

The molecular dynamics tutorial is divided into 5 modules. The first 4 are substantive and are designed to introduce molecular dynamics, design patterns, graphical user interfaces, and threading respectively. The final module serves largely to review outcomes from the previous modules, and present a much more advanced final product with a discussion of the additional considerations used in producing this final product. Below we briefly summarize the key features of the 5 modules.

3.1 Module #1 - Basic Molecular Dynamics

The aim of module 1 is to obtain a basic working molecular dynamics code. The starting point is a cubic “box of atoms” of size R and with N atoms positioned along each axis – giving a total of N^3 atoms. At this point the “box” is used primarily to provide a simple definition for the starting coordinates of each atom, but in due course

it is linked to the idea of atom packing and concepts like body centered cubic or face centered cubic structures. Integrated into the initial problem definition section of this module is a discussion of data structures, and a comparison of using a list of atom objects versus a list of vectors. The ease of interpretation associated with a list of atom objects is contrasted with the performance advantage of a list of vectors.

The tutorial then introduces an elementary Lennard-Jones interaction potential and leads the students through evaluation of the total potential energy of the system and force on each particle. With these components two options are pursued, one to minimize the structure of the system, and another to perform a dynamics calculation using a simple integrator and recording the potential, kinetic and total energy at each timestep. With these basic pieces in place the students are invited to add the few lines of VPython required to visualize the system as the coordinates change.

At this stage the students have already produced a simple molecular dynamics code and are in a position to explore some the underlying issues involved in using the code. For example they experimentally evaluate the performance of their code with respect to the various input parameters, such as timestep size, number of timesteps, number of atoms, initial atom coordinates. The behavior of the code as a function of the timestep is also studied, with the goal that the student recognizes when the results produced are no longer physically valid. The form of the interaction potential is considered, with the idea of producing an O(n) scaling algorithm by using a cutoff to only compute numerical significant interactions investigated.

3.2 Module #2 - An Introduction to Software Design

At the end of module 1 the student has produced a very simple program capable of performing basic structural minimizations and dynamic simulations on a group of interacting atoms. Module 2 posses the question of adding functionality to their code, e.g. what if we want to add an option to colour atoms according to energy, or augment the graphics to display arrows on each atom indicating the force on that atom. The aim here is to indicate to the student that without any formal methods of design it is possible that their software can very quickly become hard to develop and manage. The students are pointed to an article by Dianna Mullet [4] highlighting "The Software Crisis".

Following this preamble the student is introduced to the concept of a design pattern. Patterns are an attempt to describe workable solutions to known problems in a manner that enables these solutions to be easily applied when new "similar" problems arise. Although the concept of "patterns" and "pattern languages" dates back to the late 70's when they were introduced by Alexander in the context of building design [5], it was not until the late 80's that widespread interest was aroused within the computer science community. Since then the benefits of this approach to software design has been widely recognized, in part due to the landmark book "Design Patterns: Elements of Reusable Object-Oriented Software", that was published by Gamma, Helm, Johnson and Vlissides in 1995 [6] and lists 23 software design patterns.

While the concept and use of design patterns is now well established within the computer science and software engineering community, this is not the case within the

high performance computing community. The aim of this module is to make the reader aware of design patterns and give them a flavor for their utility in a computational science setting. It is not intended to be a comprehensive introduction to the topic. The tutorial starts by considering the model-view-controller pattern as this is one of the most commonly used design patterns in graphical applications. It separates the software into three main areas.

The model: this is the part of the software that performs the function that the software is being written for. In the case of this tutorial there are two models, a minimizer and a dynamics simulator, and the algorithms that go with them. The model provides an interface for accessing the data that it processes, and for telling it how to behave.

The View: this takes the data that the model has calculated and renders it in a format suitable for a person or another program. The format may be displaying it as graphical text, storing it on disk or transmitting it over the internet. In this tutorial the view is the VPython output.

The Controller: this tells the view anything it needs to know about how to display the model, or the model anything it needs to know about how to behave. It starts and ends the program. The controller will usually handle all user input, and send it to the models or views accordingly.

Within this environment if the user wants to print out the kinetic, potential and total energies they would create a new view, but they would only have to write this code once since both the minimizer and simulator can use the same view.

A problem with the model-view-controller pattern is dealing with more views. Each time a new view is added code would need to be added to the models to tell them to update the new view; if the user wanted to stop updating a particular view, conditional statements need to be added to enable this. A better approach is provided by the observer pattern, this completely decouples the view from the models. For the purpose of the observer pattern the models are the subjects and the views the observers. The observers are purely that, they observe a subject and don't have any impact on the subject. To facilitate this a common interface to all observers is required, then subjects can easily notify them when something has changed.

Since the observers are now responsible for knowing which subject to look at, two subject methods are required:

`Attach(observer)`: adds an observer to the list of observers

`Detach(observer)`: removes an observer from the list of observers

As well as this a new method "`notify()`" is required to inform all observers that the subject has changed. With this framework it then becomes the responsibility of the controller to create the observer(s) and appropriate subject, attach observer(s) to the subject, and finally tell the subject to run.

Given a basic understanding of the observer pattern, its implementation in the context of the molecular dynamics code is discussed. This involves defining abstract base classes for the subject and observer, with two derived subject classes for the minimizer and simulator, and one derived observer class for the renderer. As this requires the use of inheritance, the tutorial contains a brief overview of this concept

and its implementation in Python. Figure 1 contains a unified modeling language (UML) diagram showing the observer pattern as applied to the molecular dynamics code.

A variety of exercises are included to illustrate the advantages of the new modular software design. For example, the user is invited to implement another observer that simply prints the values of the total energies in a text window – a task that can now be done without making any changes to the simulator or minimizer.

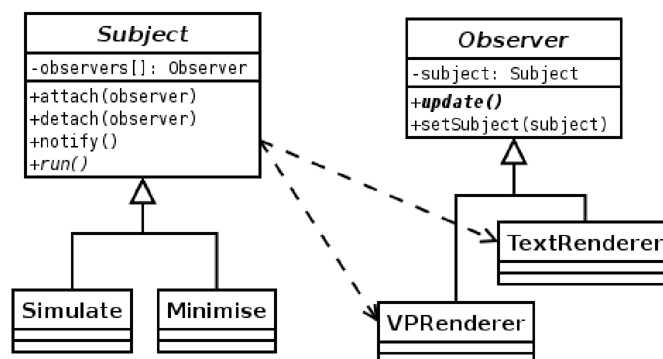

Fig. 1. The parent subject class contains all the code that handles the observers. The children, Simulate and Minimise, implement the `run()` routine, and whenever their state changes, they call `notify()`, which in turn calls update() on each observer that is currently attached to that subject. The `update()` routine will then use the publicly available data (such as the lists of atoms or forces on each atom) provided by Simulate or Minimise and render it appropriately.

3.3 Module #3 – Graphical User Interface

At the end of module 2 the basic molecular dynamics code from module 1 has been rewritten to be based on a modular object oriented design conforming to the observer design pattern. Input to the code is still provided via the command line, which with only a few parameters is not a major issue. The student is, however, asked to consider how expanding the number of input parameters soon leads to unwieldy command line input, and perhaps a better option is to use a Graphical User Interface (GUI).

In designing the GUI portion of the tutorial three possible graphics packages were considered:

Visual Python: this has its own interface package. It is very easy to use, with a simple call used to check if the user has interacted with it. Unfortunately, its functionality is somewhat lacking, in particular the only way to input a value is through the use of a slider and the user has no means of knowing the exact value. For the purpose of our simulation this was unsatisfactory

Tkinter: is the main graphics package used in Python applications [7]. It is a set of Python bindings for the Tcl/Tk graphics package. It is part of any Python distribution and as a result is easily portable

PyGTK: is a set of Python binding for GTK [8]. GTK (The Gimp ToolKit) is a relatively easy to use graphics package that was originally written for and used by the GNU Image Manipulation Program (GIMP). It has since become one of the most widely used graphics packages on UNIX platforms.

Since GTK has many more widgets than Tcl/Tk, is also faster, and since we had some previous experience with GTK this was chosen for use in the tutorial. Students who have never used any graphics package before are recommended to do the "Getting Started" chapter of the PyGTK tutorial [8].

The initial goal for the tutorial is to create a basic GUI containing a drop down menu that can select either "Minimize" or "Simulate", and four boxes where values for the number of atoms along each side of the cube, length of each side, timestep and total number of timesteps can be given. At the bottom of the GUI there are two buttons, one to start and one to quit the calculation.

Placement of the main GTK loop within the controller is discussed, as is the difference between the graphics that the VPython renderer displays and the graphics that the controller displays. Exercises are included that have the student modifying the GUI so that when the minimizer is selected the input options for the timestep size and total number of timesteps are disabled (or "grayed out").

3.4 Module #4 – Threading

At the end of Module 3 the students have not only produced a well designed molecular dynamics application code, but have also constructed a basic graphical user interface. Their attention is now drawn to the fact that there is no way of stopping or suspending the minimization/simulation once the start button has been depressed and until it has finished minimizing or run for the required number of timesteps. What if we wanted to change some parameters on the fly, such as to turn text rendering on or off, or to add arrows to the graphical output indicating the forces on the atoms? The difficulty is due to the need to run two loops at once, one is the GTK loop which waits for a widget to emit a signal and then calls any attached callbacks, while the other is the main simulation loop that updates the atomic positions. The easiest way to run both these loops concurrently is to have two threads.

The tutorial uses the Python threading module to implement the subject (i.e. either the minimizer and simulator) as a separate thread from the master thread. Sharing of data between threads is discussed, as are basic synchronization constructs and concepts like "busy wait". The student is assigned the task of adding a pause button to the controller, modifying the start button to read either start or stop, and making the program automatically desensitize the pause button when the simulation has stopped.

3.5 Module #5: The Final Product

By the end of module 4 the student has assembled quite a sophisticated computational science application code and environment. The purpose of final module is to present the student with a more advanced version compared to their final product and to provide some discussion as to how this differs from their code. In the provided code

the controller is packed full with additional features, there are more functions, and more design patterns have been applied. A screenshot of the final product is given in Figure 2.

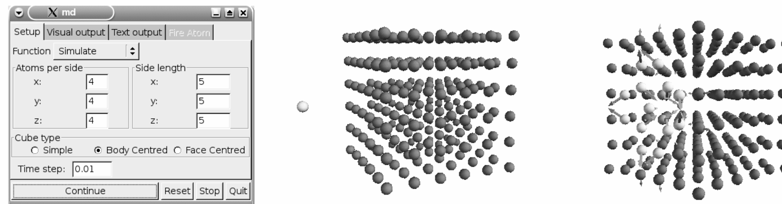

Fig. 2. Control panel and two screen shots to illustrate i) the firing of an atom into a previously minimized cube of target atoms and ii) the resulting force vectors after collision.

Some of the additional issues raised include:

Template pattern: what if we wanted to place a charge on the atoms, or define periodic boundary conditions, or in some other way modify the basic interaction between the particles in our system? This would require a modification to each subject, and then the creation of multiple sets of subjects. A better option is to use a template pattern, e.g. MDTemplate, that has several deferred methods, including force evaluation, potential evaluation and velocity update. For each type of function a class can be created that inherits from MDTemplate. To use this new implementation a method is added to the subject class, so that the subject class can transparently use whatever utility it is given.

Decorator Pattern: the provided code includes a new subject called “fire” that shoots an atom into the main cube of atoms (such as might be done in a simulation of atomic deposition). The fire subject is essentially identical to the original simulation subject, the only difference being that the fire subject adds an atom with a velocity vector that ensures that it will collide with the cube of atoms. Since fire and simulate are so similar it is more efficient to let fire inherit from simulate and then redefine or add methods as required. The end result is that fire only has the code relevant to adding a firing atom, and it does not need to worry about simulating it. The only other required modification is to the renderer that needs to be changed in order to accommodate the target and arrows. Due, however, to its well planned design none of its code needs to be altered, only the relevant parts added.

A Configuration Class: the provided code can minimize a cube of atoms and then fire an atom at the resulting structure. To enable this functionality a new “configuration” class was added to store all the information concerning the structure of the atoms. It is initially imported into each subject before running and then exported when that subject finishes.

The GUI: the final GUI is considerably more complex than at the end of module 4. As the size and complexity of the application increases it is very easy

for the code associated with the GUI to become very messy. Some discussion on how to improve the design of the GUI is given.

At the end of module 1 the students had a VPython code of roughly 200 lines. The code provided in module 5 and excluding the GTK interface contains roughly 650 lines. This is much larger than the original code, but it is now possible to write quickly a controller that will do far more than the original program, and do this without touching the exiting code. It is also possible to add new functions and incorporate different environments and forces in a relatively clean and easy manner.

4. Conclusions

In the 70's and 80's computation was successfully applied to modeling a range of physical phenomena. While much useful work was undertaken to develop the underlying methods and algorithms the associated programs typically evolved in a rather haphazard fashion, had primitive input/output capabilities, and were run on large mainframe systems with little interactivity. Today we seek to apply computational methods to much more complex problems, using computer systems that are considerably more advanced. Graphical user interfaces are considered the norm, greatly helping in input preparation and post simulation analysis. As a consequence computational science requires practitioners with some understanding of good software design, how to build and use a graphical user interface, and an appreciation for concurrent and parallel programming issues. In this tutorial we have attempted to demonstrate the importance of these skills in the context of building quite a sophisticated molecular simulation environment. VPython was found to be a useful vehicle for conveying these ideas.

Acknowledgements: The authors gratefully acknowledge support from the Computational Science Education Program of the Australian Partnership in Advanced Computing

References

1. The Australian Partnership in Advanced Computing, see http://www.apac.edu.au
2. Visual Python, see http://www.vpython.org
3. S. Roberts, H. Gardner, S. Press, L. Stals, "Teaching Computational Science Uisng VPython and Virtual Reality", Lecture notes in computer science, **3039**, 1218-1225 (2004).
4. D. Mullet, "The Software Crisis", see http://www.unt.edu/benchmarks/archives/1999/july99/crisis.htm
5. C. Alexander, S. Ishikawa, and M. Silverstein, "A Pattern Language: Towns, Buildings, Construction", Oxford University Press, New York (1977) ISBN 0195019199
6. E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley (1995) ISBN 0201633612
7. Tk Interface (Tkinter), see http://docs.python.org/lib/module-Tkinter.html
8. PyGTK, see http://www.pygtk.org/