DECLARATION OF ORIGINALITY

This report is my own unaided work and was not copied nor written in collaboration with any other person

Project Number: #79
Project Title: Algorithm Visualisation
Name: Nathan D. Pitman
UPI: npit006
Partner Name: David J. Olsen
Supervisor Name: Dr. Michael J. Dinneen
Date Submitted: 10/5/10/

INTERIM REPORT – ALGORITHM VISUALISATION PROJECT

Nathan D. Pitman

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

Abstract

This interim report discusses the Algorithm Visualisation Project spearheaded by Nathan Pitman and David Olsen. Algorithm visualisations have already proven themselves to be useful as tools for computer science education and visual debugging. The problem with existing algorithm visualisation tools is that they are algorithm-specific, offer relatively low modifiability and are functionally restricted. This report discusses in detail the algorithmic visualisation library being developed for this project. This library is intended to provide algorithm visualisation for supported languages in a generic fashion, such that the user need only reference it once in their code (and need not modify it in any other way) to be presented with a visualisation at runtime. Preliminary project goals, design, current and future implementation details, and project management details are also discussed in detail. At this stage, there is a proof-of-concept demonstration for this project which visualises the Bubble Sort algorithm operating on a C++ vector collection.

1. Introduction

1.1. Aim

The purpose of the Algorithm Visualisation Project is to provide a way to generically visualise algorithms during program execution. Specifically, the goal is to develop an algorithm visualisation library which, with minimal effort on the part of the user, can graphically depict the data structures on which algorithms operate at various stages of their execution.

1.2. Background / Motivation

Algorithm visualisation is useful in many fields of computing where visual abstraction aids understanding of code in execution. It is potentially useful to lecturers of computer science who want to provide a visual representation of specific algorithms for the benefit of their students. Algorithm visualisations of this type generally take the form of animated GIF pictures [1] or Java applets [2]. The inherent problem with visualisations like this is that they are algorithm-specific and new visualisations must be specially developed for each new algorithm which is formulated as they have a low level of flexibility and modifiability. It is not troublesome to do this for popular and well-known algorithms but lesser-known, experimental or business specific production-code algorithms are neglected in this respect.

Algorithm visualisation also has potential to be used for the visual debugging of code as the debuggers typically packaged with compilers are poor for visualising complex (non-primitive) or user-defined data structures. Furthermore, they are highly restricted in their functionality in that they only display the current state of the algorithm in execution (i.e. the data structures at discrete points) and do not allow previous states to be viewed and do not emphasize the relationship between states or the lifecycle of data. There are already visual debuggers in existence [3] but they do not allow the user to backtrack through the process of algorithm execution and tend to focus on discrete states of data structures rather than their context and the dataflow between them.

Clearly, in both of these cases, it would be useful to have a generic method of visualising algorithms, such that code could be written in a regular fashion and the programmer need simply reference an algorithm visualisation library without altering their original code in order to be presented with a visualisation of the program while it is running. That is the motivation behind this project.

1.3. Terms Used

Here is a list of terms which are used throughout this document:

- Algovis: A version of this project's Algorithm Visualisation Library.
- User: A computer programmer who is writing code and desires to visualise it using Algovis.
- Supported Language / Data Structures: If a language is described as a supported language, it means that there is a version of Algovis which is capable of visualising a subset of data structures defined in that language and libraries written for that language. These data structures are referred to as "supported data structures".

- Retrofitted Code: This refers to user code which has been linked against Algovis.
- Introspection: The ability to examine the state of a data structure at runtime (for Algovis purposes).

1.4. Preliminary Project Goals

At the commencement of this project, some preliminary goals were formulated as stated below:

- Algovis should allow user code to be retrofitted by simply referencing it so that it requires minimal hassle on the part of the user. It is a goal that the user need modify as little of their code as possible.
- When retrofitted code is executed, it should execute exactly as the non-retrofitted code would (from a functional perspective) but also provide a visualisation of any supported data structures used in the retrofitted code.
- This visualisation will initially involve displaying static graphical representations of the set of supported data structures at discrete points in time. For example, at time t_0, an array structure might contain 4 elements and the visualisation at that time should depict this. At a later point t_1, another element might be inserted, and so the visualisation would be updated to depict this.
- Later versions of this visualisation will include animations which meaningfully indicate the transitions between these static views. For example, if a piece of data is copied from one array to another, the visualisation will show this piece of data moving from between the arrays.
- Algovis should be able to maintain a history of each instance of a supported data type from the time it is created to the time it is deleted. This means that data can be tracked throughout a program which is useful for visualisation and statistics.
- Algovis should be easily extensible such that the user can add drawing capability for their own abstract data types and other unsupported data structures.
- Algovis should support code written in as many languages as possible. Furthermore, it should be architected in a fashion that support for a new language can easily be added.
- Algovis could possibly also provide statistical data on each supported data structure as well as visualising it.

2. Design and Project Management

This section details major design and project management decisions made in the course of this project

2.1. Software Architecture

It was understood from the beginning that there would be two major tasks involved in developing Algovis. The first task is to be able to introspect all changes to supported data structures at runtime and use this information to record the dataflow for that data structure. The second is to be able to provide a meaningful visualisation of this introspected information.

The decision was made to perform these two tasks in separate processes such that there would be a number of different versions of the Algovis introspection component (one version for each supported language, most likely written in the supported language) and one version of the Algovis visualisation/statistics component. Ideally, the decision to support a new language in Algovis could be accomplished simply by developing another Algovis introspection component which caters for the data structures the user desires to support in that language. This introspection component (in its own process) would communicate with the visualisation component (also its own process) via IPC (Inter-Process Communication). This design mitigates the risk of a developer attempting to support a new language only to find that it is infeasible, as no time is spent developing language specific visualisations.

Essentially, this architecture is MVC (Model View Controller) in the sense that there is a Model introspection component for each supported language, and View components (i.e. Visualisation and Statistics views) which theoretically work with any type of Model component. By separating the introspection from the visualisation, the risk of wasting developer time exploring feasibility for supporting languages.

2.2. Platform and Language Decisions for Visualisation Component

Given that the team members have some experience in computer graphics programming, it was decided that the visualisation component would not be a major challenge to develop. Furthermore, the modular architecture allows any sort of language to be used for this component, provided that it conforms to a standard IPC interface.

The visualisation component is being developed using C++ and OpenGL. C++ was chosen as both team members are familiar with it, it is well enough established to have many libraries written for almost any application (including many graphics libraries),

powerful and versatile enough such that users are not overly restricted by language design decisions, portable across platforms, and highly efficient.

OpenGL was the C++ graphics library of choice as both team members are familiar with it, it is open-source, it offers all of the functionality required and it is cross-platform.

By using the C++/OpenGL combination, it is ensured that the visualisation component of Algovis will be usable on Windows and UNIX systems.

2.3. Platform and Language Decisions for Introspection Component

It was decided that developing an introspection component to support a language would be the most difficult part of the project implementation. It is required that every change to a supported data structure can be introspected and communicated to the View components by IPC. Bearing in mind that the set of intended supported data structures would, at a minimum, include all primitive types for the supported language as well as commonly used collection data structures, it is apparent how difficult this could be for some languages.

In general, there are a few approaches to introspection which can be used for code written in different languages and these are listed below:

- Some languages support runtime reflection of code which allows data structures to be programmatically explored at runtime. The disadvantage of reflection is that it is not necessarily possible to detect when a supported data structure is created or changed.
- Another approach involves developing a tool which works like a debugger by analysing compiler generated debug symbols. The disadvantage of this approach is that high level code is being analysed at a low level in an endeavour to recreate a high level picture and this roundabout method is not guaranteed to work.
- Another approach involves developing a tool which alters the original code in a way that allows introspection but doesn't affect its functional characteristics. These often work by parsing source files or binaries.
- Another approach which works for OOP (object-oriented programming) languages involves developing an adapter type for each type which should be supported by Algovis. The role of these adapter types is to encapsulate the original behaviour of their supported types but also to subsequently perform the introspection and IPC communication required for Algovis to work. The advantage of this approach is that it is possible to discern the exact timing and nature of any changes made to a supported data structure. One way to do this is to use a class adapter pattern (adapter by inheritance) where the adapter type inherits from the base supported type and overrides every method (including all defined operators) offered by the supported type. These overriding methods would call their base class equivalents before performing Algovis specific code. In order to ensure that these adapter types are used instead of their base types, some sort of type redefinition for each type is required. This does not work in practice because most OOP languages do not permit inheritance from primitive types.
- A variation of the above is to use an object adapter pattern (adapter by encapsulation) pattern where an object of the adapter type encapsulates an object of the supported type. An adapter object acts as a proxy by forwarding all method invocations to the encapsulated object and performing any Algovis specific code before returning from the method body. Like the above, this also requires the language to offer type redefinition to ensure that the adapter types are used.

  Java was considered as a supported language but was soon rejected as it does not allow type redefinition or operator overloading. C# .NET was more promising as it supports operator overloading but not to enough of an extent for the adapter approach to be useable. C++ supports type redefinition and full operator overloading which makes it an ideal candidate for the adapter approach. Python is an interpreted language and so it should be fairly trivial to perform type redefinition. There are Python libraries which allow for the simple construction of wrapper types [4] and this makes it another ideal candidate for the adapter approach.

3. Current Implementation, Future Work and Project Management

3.1. Current Implementation

This section details the current strands of development in this project. Sections 3.1.1 to 3.1.3 discuss three different approaches to introspection and Section 3.1.4 discusses the visualisation component.

3.1.1. C++ Wrappers Approach to Introspection

The C++ wrappers approach uses the object adapter pattern (adapter by encapsulation). C++ allows type redefinition using the #define macro which even allows primitive types to be redefined as class types. This means that by developing wrapper types for each supported type (i.e. an IntWrapper which wraps an integer type) and a header file which redefines supported types as their wrapper types, the user need only include this header once and link their code against Algovis in order to use it.

Having explored the feasibility of this option by developing and unit-testing wrappers for most of the primitive types and the STL (C++ Standard Template Library) vector type, and having tested Algovis on a Bubble Sort implementation, only two problems have been encountered. Redefining primitive types as wrapper types (classes) works well in most cases except when the C++ compiler explicitly requires primitive types. One scenario of this occurs when using non-type template parameters which must be primitive [5]. The second problem relates to the use of inbuilt C arrays. Because they are not types referred to by single tokens but rather, inbuilt language constructs, they cannot be redefined as wrapper types.

3.1.2. Valgrind Approach to Introspection

Valgrind is a framework which is used for writing program analysis tools such as memory leak checkers and cache profilers. It works by translating user code into IR code (Intermediate Representation) and performing dynamic binary instrumentation on it (injecting code at runtime).

The second strand of development has involved developing a Valgrind tool (i.e. a tool which uses the Valgrind framework) in order to introspect supported data structures at runtime. The advantage of Valgrind is that it is theoretically language independent. One disadvantage is that IR code is often too low level to be useful. Another disadvantage is that Valgrind tools cannot be linked against the C Standard Library which limits their use.

3.1.3. Python Wrappers

No progress has been made on developing Python wrappers beyond basic feasibility investigations.

3.1.4. Visualisation Component

At this stage, the visualisation component is capable of graphically depicting iterable data structures at discrete points in time as collections of quadrilaterals. Currently the visualisation component is restricted to generating primitive 2D depictions of data structures. Depicted in Figure 1, is one frame of a visualisation of Bubble Sort operating on a C++ vector 10 elements in size. Note that the C++ wrappers introspective component was used to provide the visualisation component with the data.

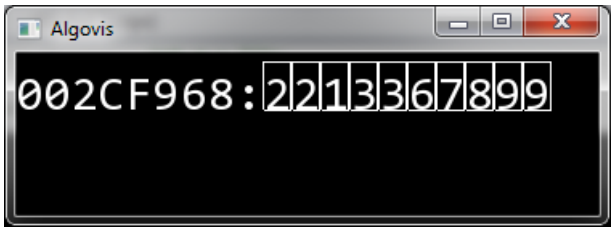

Figure 1: One frame of a visualisation of Bubble Sort

3.2. Future Work

Future plans for this project involve further developing the visualisation component and C++ wrappers introspection component and commencing development on the statistics component and Python wrappers introspection component. The visualisation component will be improved so that it shows animation between states, is aesthetically superior, and is more user-customisable by allowing the user to drag and suppress individual data structures. The C++ wrappers component is quite promising and wrappers will be developed for the remaining primitive types, STL collection data structures and for data structures of a yet to be chosen C++ graph library (in order to allow visualisations of graph structures). While the Valgrind approach is proving to be a bit difficult to implement, it is not being abandoned yet. Rather, its feasibility will continue to be explored as it could possibly be combined with the C++ wrappers approach should it be required for its language agnostic nature in a scenario where the C++ wrappers do not suffice. The Python wrappers approach is also very promising and development will begin on that as soon as possible.

Of lesser importance is a statistics View component which would display useful statistics about various data structures. For example, useful statistics for an array list might include a distribution of positional element accesses, typical read patterns as well as the frequency and nature of memory reallocations which could indicate a poor resizing policy. This component is secondary to the premise of this project and will be developed if time permits.

3.3. Project Management

This project does not conform to a strict software development process but prototyping has been successfully employed. The rationale behind using the

prototype methodology is that the project timeline can be divided up into iterations of varying size, the deliverables of which are specified according to different levels of requirements. As an example, the first iteration required that the visualisation component satisfy the requirement for static graphical representation at discrete points. A later iteration will require that the visualisation component satisfies the animation requirement.

Up until now, it has been quite simple to divide the project workload by virtue of the modular architecture and the independent strands of implementation. David Olsen has been working primarily on the Valgrind introspective component and the visualisation component. Nathan Pitman has been working on the C++ wrappers introspective component and testing. Future work will also be easily divisible in this way, and it is anticipated that both team members will contribute to all components.

4. Conclusions

This project is progressing very smoothly and the work to date shows that it is certainly feasible to develop a generic algorithm visualisation library which meets the preliminary goals. Already, there is a proof-of-concept demonstration for this project which visualises Bubble Sort being performed on a C++ vector collection. From here, support for more data structures and even more languages will be added so that Algovis can evolve into a tool which is useful for developers, whether they are novices or professionals. By the end of the project, it is intended that the Python and C++ wrapper components will be in a release state and both will allow primitive types, user-defined types, collection types and graph types to be visualised.

5. Acknowledgements

I would like to acknowledge our project supervisor, Dr. Michael J. Dinneen, for his useful guidance and perspective on this project.

6. References

[1] “Merge Sort – Wikipedia Article” retrieved on May 10, 2010 from http://en.wikipedia.org/wiki/Merge_sort

[2] “The xSortLab Applet” retrieved on May 10, 2010 from http://math.hws.edu/TMCM/java/xSortLab/

[3] jGrasp home page retrieved on May 10, 2010 from http://www.jgrasp.org/

[4] Python Package Index – ProxyTypes retrieved on May 10, 2010 from http://pypi.python.org/pypi/ProxyTypes

[5] IBM Linux Compilers – Non-type Template Parameters retrieved on May 10, 2010 from http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=/com.ibm.xlcpp8l.doc/language/ref/non-type_template_parameters.htm