Program Visualisation tool for teaching programming in C

Mr. Stephen Kirby, Mr. Benjamin Toland, Dr. Catherine Deegan
Department of Engineering
School of Informatics and Engineering
Institute of Technology Blanchardstown, Dublin, Ireland

ABSTRACT

This paper presents a visualisation tool for novice C programmers. It is well known that programming is perceived to be difficult among novice learners. The aim of this tool is to graphically visualise the code that the student is working on to give them a coherent computational model. It is hoped that this tool will help the learner achieve a coherent mental model in common with the instructor and the rest of the class. This could allow valuable class time to be concentrated on problem solving issues earlier in the teaching semester.

Keywords: programming, novice programmers, Jeliot-C, Software Visualisation, Program Animation

INTRODUCTION

With the introduction of C++ and Java to the programming world, a lot of programmers have moved away from the C language but C remains important in the Engineering industry. Most program visualisation tools currently available for the C language deal with algorithm visualisation (Figure 1) and ignore fine grained program visualisation. For novice programmers the main emphasis is on basic programming skills such as variable declaration, functions and pointers and not on sophisticated algorithms such as sorting. This project aimed to produce a program Visualisation tool for the novice programmer in C.

Figure 1. Pancake Sorting Algorithm Visualisation

The reason that this new tool has been developed for C and not C++ is twofold. One reason is that the C computational model is simpler than the C++ model which includes all of the C concepts plus additional object oriented capacities. The second reason is that C is often used as the introductory language in engineering programming – as is the case in the authors' institution.

Program visualisation. Program Visualisation involves presenting the learner with a visual representation of an executing computer program. It has been the experience of the authors that many learners struggle with programming as they do not have a coherent mental model of how a basic program executes. By allowing beginner programmers to observe a graphical representation of their program in execution they are greatly helped in developing this computational mental model and thus the learning of programming is simplified. It has been found that Program Visualisation tools help in the teaching of the Java[1] programming language but there is no easily available such tool for the C language which is still a major language used in Computer Engineering education.

JELIOT

Jeliot is a program visualisation tool developed in the University of Joensuu, Finland, which visualises Java code at a fine grained level aimed at beginner programmers. Figure 2 shows Jeliot, with its rich visualisation engine, which highlights the part of the program that is being currently executed.

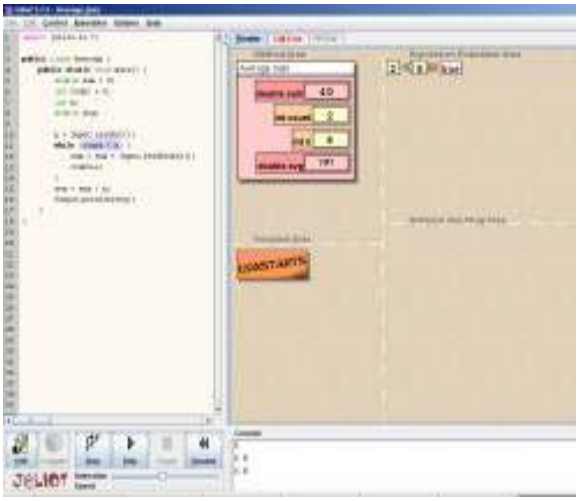

Figure 2. Jeliot

Jeliot has been designed with modification in mind so that the animation engine component can be used to animate other programming languages. To achieve this, the Jeliot team have used a novel technique to decouple the interpretation and execution of the source language (Java) from the visual animation engine. The key to this technique is a new intermediate language called MCode. The source language interpreter interprets the source code and outputs a trace of the program execution expressed in MCode. This MCode trace is then fed into the visualisation engine to produce the animation. Thus the Jeliot team hoped that others could replace the source code interpreter with a different one, e.g. a C++ interpreter that would also produce MCode and so not need to modify the animation engine part for a new language.

The Jeliot team used a 3rd party command line Interpreter (Dynamic Java [3]) as the basis of their source language interpreter. This interpreter was modified to produce an MCode trace of the program execution including all runtine information. This MCode trace was then fed into the visualisation engine to render the animation on screen. Figure 3 illustrates the functional structure of the Jeliot program [2].

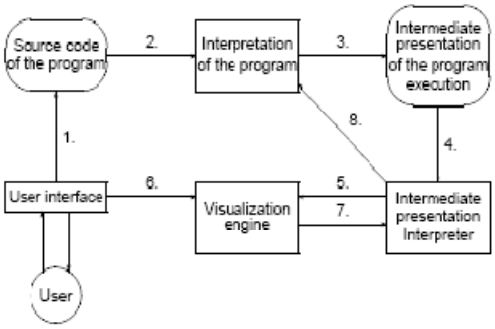

Figure 3: Jeliot Functional Structure

INITIAL PROJECT SCOPE

Thus the initial idea of this project was to find a third party C interpreter, modify it to produce MCode and simply replace the Dynamic Java interpreter. The first problem with this plan was finding a suitable C interpreter. As Jeliot is itself developed in the Java language a C interpreter written in Java was sought.

VISUAL INTERPRETER

Visual InterPreter (VIP) [4] is a program visualisation tool for a subset of the C++ programming language called C--. VIP was developed in the University of Tampere, Finland. Figure 4 shows VIP in operation. As can be seen, VIP is not as visually appealing as Jeliot. However, it does at its core have a C-- interpreter which could possibly be modified to suit the purpose of this project.

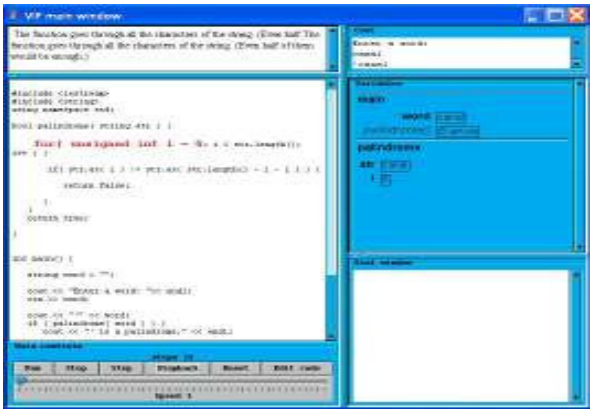

Figure 4. VIP

VIP is based on a C-- Interpreter developed in Tampere called Command Line InterPreter (CLIP). CLIP [5] is built using a “compiler-compiler” tool called SableCC [6] which generates fully featured

object-oriented (Java) frameworks for building interpreters. Frameworks include intuitive strictly-typed abstract syntax trees (AST) and tree walkers. This is the basis of how MCode can be created in VIP; as a tree walker walks across the AST, MCode can be produced and fed into the animation engine that can be viewed by the novice programmer. This is the approach taken in this work.

The subset of C++ implemented by CLIP, C--, falls between C and C++. As such it is neither a full C++ interpreter nor a full C interpreter. Key C concepts missing from CLIP include global variables, standard I/O functions and bit operators.

JELIOT-C

The new tool developed is called Jeliot-C as it takes advantage of the animation engine of the Jeliot system and the interpreter of the VIP project to produce an animation tool for C.

MCODE

MCode is an intermediate language developed by the University of Joensuu to allow them to decouple the components of their Jeliot program.

To get a better understanding of MCode an example is shown below. We begin with a simple Java statement:

```
int y;
```

For the above statement in Java the following MCode is produced by the interpreter [7].

```
VD, y, 0, ?, int, 1, 4,3,4,7
```

Where the various fields refer to the following:

```
VD, Name, NO_REFERENCE/ir,
      value,  type,
   FINAL/NOT_FINAL, Loc
```

"Name" is the name of the variable. In this case "y".

"NO_REFERENCE/ir" is a field used in complex statements to refer multiple lines of MCode to one source line.

"Value" is the value of the variable if known otherwise a ? will be inserted.

"Type" is the type of variable. Example of types include int, double, char

"FINAL/NOT_FINAL" is whether the value is final or can be altered later on.

"Loc" is the location of the statement in the source code document.

MCode was designed with Java in mind and so while MCode can express any of the concepts found in Java, and so by extension many other object-oriented languages, it cannot express some of the concepts found in procedural languages such as C and the non object oriented concepts of C++. These include:

1. Pointers
2. Structures
3. Global variables

To address this problem the MCode language had to be modified/extended to allow the use of these concepts. Below is the same example as above but expressed in the new extended MCode that supports the C language.

```
VD, y, 0, ?, int, 1, 4,3,4,7,
          0x0054
```

Where the various fields refer to the following:

```
VD, Name, NO_REFERENCE/ir,
value, type,
FINAL/NOT_FINAL/
GLOBAL/NOT_GLOBAL/
STATIC/NOT_STATIC,
Loc, Address
```

There are two fields to look at which are:

"FINAL/NOT_FINAL/GLOBAL/NOT_GLOBAL /STATIC/NOT_STATIC" is an extended field that now includes not only whether the variable is a constant (FINAL) but also whether it is a global variable or is declared to be static. This field can be any combination of the possible values.

"Address". This is a new field inserted into the declaration to include the physical memory address of the variable. This will allow the use of pointers in the animation engine to be implemented and

possibly a “memory view” animation showing the physical locations of the variables in the program.

These alterations to MCode have been agreed with the team from the Jeliot project.

The alterations to the MCode specification was the first modification required to the Jeliot architecture. Afterwards it was necessary to change the visualisation engine of Jeliot so that it could accept and visualise the new MCode. For example Figure 5 shows an initial mock-up of how it the visualisation engine would visualise pointers and structures.

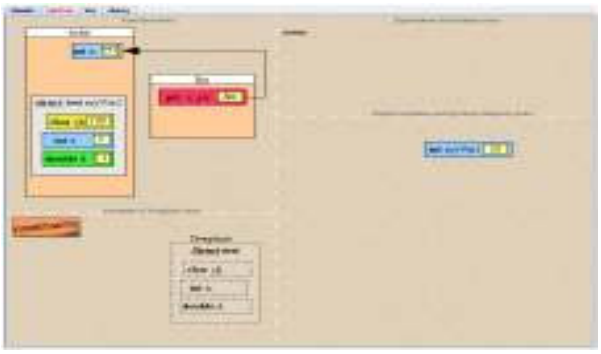

Figure 5. Visualisation of Pointers and Structures

IMPLEMENTATION

As mentioned previously MCode was designed with Java in mind and therefore it had not been designed for other programming languages that are not object orientated, such as C. This meant that the MCode language had to be modified/extended to include other concepts that are in the C language.

In implementing the project an attempt was made to extend rather than simply modify the CLIP interpreter. This would hopefully reduce the work required to re-integrate any new future versions of CLIP released.

CLIP is designed to implement C-- a subset of C++ which as mentioned previously does not fully implement all of C’s features either. This posed the question of whether the attempt to only extend CLIP would in fact be feasible. At the time of writing the authors were still adopting the approach of extending rather than modifying CLIP

The animation engine of Jeliot had been designed with Java in mind. Because of this it could not represent C concepts such as global variables, pointers or structures and so additional animation capabilities had to be added. The animation engine also had to be updated to be able to accept the new MCode specification. Again, an attempt to extend rather than modify the animation engine code was made to try and reduce the re-integration effort required if a new version of Jeliot were to be released in the future.

CONCLUSIONS

According to a current research project [8] in the authors' institution analysing the learning styles of incoming students in engineering and computing 71% have been found to be visual learners. Visual learners are students who prefer the use of visual techniques, such as diagrams and colour to aid their learning. Given these statistics the authors believe that this tool will be of great benefit to many novice programmers.

A fully working prototype of the tool has been developed and will be trialled in the Autumn semester of 2010. These trials will be conducted on novice programmers and will survey the students as to their preference to use of the tool to learn a new concept as opposed to the traditional method. The Jeliot team have done similar trails on students [1] and although they have not found definitively that the tool has an impact on exam performance, it was found that the students who did use the tool found it useful. Students who regarded themselves as weak or who struggled in programming reported most favourably on the use of Jeliot.

The authors expect similar results in the trialling of Jeliot-C with novice Engineering students. While no quantitative measure of improved examination performance is expected (or will be measured) it is hoped that both learners and instructors will report benefit through the use of a highly visual and common mental model.

REFERENCES

[1] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, Pekka A. Uronen The Jeliot 2000 program animation system, Pub 2002

[2] Niko Myller, The Fundamental Design Issues of Jeliot 3, Masters Theses, Pub 2004.

[3] Koala Project, 2002. DynamicJava. WWW-page, http://old.koalateam.com/ (Accessed 16/12/2009)

[4]VIP, 2008, WWW-page, http://www.cs.tut.fi/~vip/en/, (Accessed 16/12/2009)

[5] CLIP, 2008, WWW-page, http://www.cs.tut.fi/~vip/clip/clip_english.html, (Accessed 16/12/2009)

[6] Etienne Gagnon, SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, Pub 1998.

[7] Andrés Moreno García, The Design and Implementation of Intermediate
Codes for Software Visualization, Pub 2005

[8] G. Gray, D. Duffin 2009. Learning Styles Sub Theme, *ContinueIT: Strategic Innovation Fund.* Project Report: 2007-9. Available on request from geraldine.gray@itb.ie. Unpublished report.