Literature Review: Visual Programming System

Matthew van Cittert

1 June 2009

1 Introduction

1.1 Overview of project

This project hopes to recapture the sentiment that there may better ways to program computers than those in widespread use. Modern Integrated Development Environments (IDEs) provide a number of tools and facilities to aid navigation and editing of text files. But these follow the common trend of providing a central text editor to make changes at the character level, and various tools about the periphery to describe and navigate the codebase under inspection.

This project intends to take a slightly different approach. Using principles and ideas from the fields of visual programming and human computer interaction, the aim is to pose a shift of emphasis from character level editing to a top-down graphical representation of a program's structure.

To function fully, such a system would require a number of core components, some of which form part of the research and others which are no more than mechanical requirements. The most central and important component, from a research perspective, is the interface. As the aim of the project is to provide a novel interface to code, the bulk of this review will be centred on designing and evaluating the interface.

The second required component is an intermediate format, from which the interface may query information about the code in order to create representations. A possible format to use is that of an ontology. Such a route would allow ad hoc queries to generated, and shift the responsibility of answering these queries onto an existing reasoner. But use of ontology is not central to this project and remains an optional design decision, to be pursued only once sufficient progress has been made on the interface.

The final most mechanical component is a parser. This is required to handle transformations between textual code and the intermediate format. No research will be done in the field of parsing, and no literature on parsing is presented. As the parser does not contribute to research, it is possible to omit implementation of a parser completely, and still demonstrate the interface using handcrafted data in the intermediate format.

1.2 Related fields

Visual programming was a major field of study between the mid-eighties and mid-nineties [1]. As computer hardware advanced, so did the ability to render and display graphical content. The proponents of visual programming hoped to harness this technology, and so advance the techniques used to program computers from simple text to more natural methods. The aim of this endeavour was to improve the translation between deriving and describing the solution and encoding it in a programming language [2].

The field of visual programming encompassed a number of different areas of research. Some researchers aimed to produce graphical front-ends for existing textual languages. Others created entirely new languages based on diagrams and graphics. Some abandoned textual encoding completely and spatially parsed pictures and graphics as code.

This project intends to provide a high level front-end interface representing the data structure of existing textual languages, but leaving the low level implementation of algorithms as text. This excludes much of the research in visual programming, including graphical representation of algorithms, derivation and parsing of new graphical languages and the use of visual programming in education. Despite this, some of these papers have presented interesting questions, arguments or observations relevant to the current study, some of which are presented here.

Other more relevant papers have focused on issues such as how users perceive and interact with programming languages, considerations to take into account when designing interfaces to programming languages and means of evaluating such interfaces. The papers reviewed centre on the work of [3] and [4].

2 Contributions from existing systems

2.1 Introduction

A wealth of visual programming systems have been created, introducing a number of different ideas and designed to fulfill a number of different aims. Some are front-ends for textual languages, some are languages designed to take advantage of graphical representations. Some are meant to make programming easier and more accessible to novices or non-programmers, some service specialised fields such as audio or electronics, some focus on data structures, others focus on algorithms. This section introduces some of the ideas and arguments that factored into the design of various systems, and some comments or criticisms of various ideas and design decisions.

2.2 Pict [5]

Pict was a data flow oriented visual programming language designed to run without any use of text or the keyboard, the paper's unsubstantiated claim having been that graphics are inherently better and easier to process than text. Instead, [3]'s view that both text and diagrams each have different strengths and weaknesses is a far more realistic view. Establishing when and how to use text or graphics is essential before embarking on the design of any visual system. This is explored in the next section.

Another claim by [5] is that visual systems are suitable only while learning to program, and that users should make the transition to textual programming as they gain expertise. This is in contradiction to the conclusion drawn by [4] that the extra room for expression provided by visual languages makes their use more complex for novices, and that expertise is required to use this room for expression in a beneficial way. Despite [5]'s view, this project intends to address some of the issues of experienced programmers using ideas from visual programming, with the view that only a relatively short period of a programmer's career is spent learning to program.

But a valid point raised by [5] is that graphical symbols may be used to compress a lengthy textual description into a compact graphic. But [4] has raised the equally important point that the interpretive value of images is not always beneficial, but can lead to ambiguity. With this in mind, it possible to use graphics effectively to express large amounts of information in a limited space, but that the meaning of such graphics must be clearly defined to avoid ambiguous interpretation.

Some interesting ideas demonstrated by [5] were the use of graphical animation to show code execution for debugging purposes, and the ability to compile and run incomplete sections of code, to allow design and testing without writing a full program.

[5] evaluated their system using an opinion survey. A number of issues about these tests were raised in the paper, such as the role novelty of the system may have had, and the presence of Pict's author during testing. A number of alternative methods of evaluating systems are presented in the next section.

2.3 Tinkertoy [6]

Tinkertoy was a graphical data flow oriented front-end for Lisp. An important observation made was that users will not work with anything more difficult to use than what they currently have. This highlights the importance of evaluation and comparison with existing means of programming.

A valuable contribution arising from the added interface between the user and code, is that systems such as Tinkertoy ensured syntax mistakes could never occur. Tinkertoy also used the powerful concept of context-sensitive menus attached to icons.

But it also highlighted that visual representations of languages may become very awkward when over complex. It is this issue that features partly in the decision to leave routine level implementation of the current system as text.

2.4 ThinkPad [7]

ThinkPad was a data structure oriented front-end developed for prolog. In ThinkPad, users were allowed to choose between a circle and a rectangle to represent a data structure. This would allow programmers to use different styles for different programs, and the lack of consistency could make reading such representations difficult [3].

A criticism of both ThinkPad and Tinkertoy is raised by [8]. This paper claims that visual front-ends should completely hide the textual representations they abstract. But this seems unnecessary for two reasons. Firstly, users experienced with textual languages could use the textual representations as guides. Secondly, as each representation may be better or worse at expressing different concepts [3], the textual representations may be helpful where graphical representations are ambiguous, complicated or do not sufficiently emphasise the required information.

2.5 Prograph [2]

Prograph was a data flow oriented system. The paper claims that one of the advantages of graphical representation is that, for example, a variable of a bicycle could be represented as a bicycle. But this recalls the argument by [4] of interpretation - all too often such representations are more easily understood by reading the textual captions. Other issues involved are the time , inclination and secondary tools required to draw the pictures for the multitude of, sometimes very temporary, variables appearing in a typical project. Finally, not all variables have an associated graphical representation. To avoid these issues, it may be better to use a limited set of commonly used graphics, the meaning of which users are required to learn.

An interesting idea raised by [2] is that graphical representations should act in a way with which users are familiar. For instance, double clicking on a class should open a window to view its contents, analogous to double clicking on an operating system folder. This metaphor could be extended in a number of ways. For example, file extension could be used to represent variable or method return type, and shortcuts used as pointers. Read and write access rights could represent the availability of get or set methods. Likewise, these virtual folders could be dragged, copied or renamed using the same shortcut keys or techniques used on operating system folders.

3 Interface design and evaluation

Before design a system, it is important to be aware of the end goals, and to ensure enough time is set aside for evaluation. To estimate this time requires some idea of what evaluation needs to take place. [3] and [4] provide a number of insights and observations important to the design and evaluation of programming interfaces. Ideas from both of this papers will be discussed in reference to the design of the project interface.

3.1 Text and graphics considerations

[4] raises issues on the uses and roles of text and graphics. These arguments, and there relevance to the project are discussed below.

3.1.1 Interpretation

[4] has given a comprehensive overview of the issues relating to graphical programming and the setbacks that have prevented its dominance over textual programming. He poses that text is more suited to programming, as programmers need a precise, almost mathematical, notation to unambiguously instruct the computer how to go about a task. This is opposed to graphics, whose ability to convey multiple and complex concepts using limited space and complexity is due to the interpretation afforded by pictures. A problem with graphics is then the issue of ambiguity, that different people may interpret the same picture in different ways. This is valid but ignores an important issue. Words are also susceptible to ambiguity and to interpretation according to the context in which they are used. Many sentences or texts may likewise be interpreted in many different and often completely opposite ways. The words used in programming

languages are restricted and strictly defined for that language. In the same way unfettered graphics are open to interpretation. To use graphics in programming, the images would likewise need to be restricted and their usage and meaning strictly defined.

3.1.2 Visual cues

An important part of programming emphasised by [4] is that of layout, termed secondary notation. The readability of code is greatly enhanced or crippled by the appropriate or inappropriate use of layout. Indentation, spaces and conventions all help convey the meaning of textual code and was found by [4] to be a major distinguishing factor between novice and experienced programmers. With graphical programming layout becomes an even greater issue due to the departure from text to a greater reliance on visual cues. For a graphical programming language to ease interpretation, focus would need to given to encouraging or enforcing a logical layout of graphics.

3.2 Interface design considerations

[3] has raised a number of issues and insights important to the design and evaluation of programming interfaces. These range from the intended audience and purpose, to accommodating the style of programmers. These points or questions are discussed with reference to the system design and evaluation.

3.2.1 Who is the audience and for what will they use the interface?

In this project, the interface is meant to facilitate general programming, and that in existing textual languages. Initially, these would be procedural languages, but the interface should not be constrained to only these.

The wider the domain demonstrated and the larger and more complicated the projects handled gracefully by the system, the more capable the system and likewise the more successful. Current IDEs are more than capable of efficiently navigating small projects, but it is large projects using a multitude of files and interacting classes that pose a problem. The aim of the interface is to improve on this situation. A quick judge on the success or failure of the project is then its ability to handle very large projects. Users of these projects should be able understand such projects, or portions thereof, more quickly than is currently possible. Likewise, maintainers of such systems should be able to update, extend and trace errors more efficiently.

3.2.2 What activities should the environment facilitate?

In this system, the aim is to allow rapid navigation, exploration (without getting lost), and top-down editing. Design and evaluation of the system should focus on these activities.

3.2.3 Deciding on a notation

The paper raises an important point, namely that while some notations emphasise certain pieces of information, they do so by hiding others. The choice of whether to use text or diagrams, and which diagram, is very contextual and depends on the information to be conveyed.

Rather than attempting to find a single all-purpose notation, or midway compromise, it may be better to use multiple views, each emphasising certain aspects of code. The user may then decide which is appropriate to his style and to the issue at hand.

3.2.4 Spatial reasoning

When searching for information, users may use spatial orientation to find it, for example remembering that some item was last in the bottom right corner of the screen.

This concept is central to the design of the project. By presenting a single workspace, necessarily larger than the screen, the spatial reasoning of user is given full support. By allowing or omitting only this component, for example exchanging the single workspace for multiple pages in a tabbed interface, the importance of spatial reasoning to the project could be assessed.

3.2.5 Style of development

Not all programmers are the same. Not all projects are the some. Some users or projects are more disposed to top-down editing, others to bottom-up editing, and there will more than likely be a mixture of both.

When designing the interface, users should not be constrained to a predefined style. Neither should they be hindered from changing styles. Ideally, both top-down and bottom-up editing should be possible from the same view, to avoid the user having to re-orient himself at every switch.

When evaluating the interface, it may be important to use test projects that are disposed to either style, or a mixture of both.

3.2.6 Workspace

Programmers generally write in small spurts, iteratively writing a bit here and something else there, then combining it all together again. It may be a hindrance to users if they have to flip between multiple pages during this process. Rather the window or workspace needs to be large enough to allow the user to jump between different parts of the project. Alternatively, they should be able to bring the different pieces together so that they fit into the limited workspace. A similar situation arises where users search or browse information, such as to check dependencies make comparisons.

Again, a workspace larger than the screen addresses partially addresses this issue. But it may also be beneficial to create temporary mini-workspaces, where users can add or remove different views and components, then either delete or store this assembly for later reference.

For evaluation purposes, tests should include this component, encouraging testers to jump to and from various parts of the project.

3.2.7 Debugging program fragments

To avoid bugs, programs are often written progressively by writing a small fragment, then testing it. Likewise, when debugging code is often pruned into small segments by commenting out surrounding code, to isolate the faulty piece of code.

As debugging is a major part of programming, it is important for the interface to facilitate this process. This could be aided by making debugging simpler, such as commenting out entire structures at a click or automatically commenting out all dependant code as well. Systems such as Pict [5] allowed the programmer to interactively and visually debug a program, and to run incomplete sections of code which halt when they run out of code to execute. Similar facilities may be replicated by integrating the environment with a debugger, but may not be as effective if routine level implementation remains textual rather than graphical.

Evaluating this portion of the interface could be achieved by testing the rate at which a test project is debugged.

3.2.8 Code reuse

Programmers often reuse old code, possibly modifying it in some way. Code is copied and modified by users attempting to learn from an example, or using it as base for further development.

The interface should allow users to import, export, copy, integrate and remove portions of code with minimal hassle. Allowing the user to interact with the code at different levels, such as entire classes or methods, should improve on the character level interaction provided by most text editors.

For evaluation, test projects may involve use and integration of existing fragments of code.

3.2.9 Following paths through code

Different environments provide different means to browse through code. But following a path through code, and importantly navigating back along this path, is often complicated. For instance, some editors allow the user to follow hyperlinks. But returning to the hyperlink may be difficult, and so IDEs often provide bookmarks. Bookmarks may also be used in the above example of visiting separate parts of a project.

Providing a single large workspace should alleviate this issue, especially if a minimap is provided to allow global scale navigation. In place of bookmarks, beacons or landmarks could be placed to remind users of where they are and where they wish to be.

For evaluation, test projects could require intricate navigation.

4 Summary

A number of visual programming systems have been developed in the past. Not many of these were used in mainstream software development. But their development nonetheless generated many interesting ideas, from which this project could draw a wealth of inspiration. But before designing such a system, it is important to have an appreciation of the issues involved, and a set of goals towards which development is aimed. Insightful papers such as [3] and [4] provide many arguments and interesting points for consideration. Constantly aiming toward these goals, and carefully avoiding the mistakes and pitfalls of the past, will hopefully yield more successful techniques of programming than those currently available.

5 References

References

[1] B. Myers and A. Ko, "The past, present and future of programming in hci," *Human-Computer Interaction Consortium (HCIC'09)*, 2009.

[2] P. Cox, E. Giles, and T. Pietrzykowski, "Prograph: A step towards liberating programming from textual conditioning," in *1989 Proceedings of the IEEE Workshop on Visual Languages.* Washington, DC, USA: IEEE Computer Society Press, 1989, pp. 150–156.

[3] T. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174, 1996.

[4] M. Petre, "Why looking isn't always seeing: readership skills and graphical programming," *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, 1995.

[5] E. Glinert and S. Tanimoto, "Pict: An interactive graphical programming environment," *Computer*, vol. 17, no. 11, pp. 7–25, 1984.

[6] M. Edel, "The tinkertoy graphical programming environment," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1110–1115, 1988.

[7] R. Rubin, E. Colin, and S. Reiss, "Think pad: A graphical system for programming by demonstration," *IEEE Software*, vol. 2, no. 2, pp. 73–79, 1985.

[8] G. Rogers, "Visual programming with objects and relations," in *1988 IEEE Workshop on Visual Languages.* Washington, DC, USA: IEEE Computer Society Press, 1988, pp. 29–36.