HYPER STRUCTURES + VISUAL PROGRAMS

=

HYPERPROGRAMS

A thesis presented in partial fulfillment of
the requirements for the degree of Master
of Science in Computer Science at
Massey University

by

Craig Robert Simmons
March 1994

Preface

This development described herein - the first implementation of a hyperprogramming language - is related to, but far more wide-reaching than my original masterate project, which was to develop an editor for structure diagrams. Early in the course of the research, one of my supervisors, Paul Lyons, invented the hyperprogramming concept and designed a preliminary version of HyperPascal. In view of the fact that this original "flash of inspiration"was not mine, and the subsequent close cooperation during language development between me and my supervisors, it has proved difficult to determine the true originator of various ideas. Furthermore, as there are no earlier detailed descriptions of the principles of hyperprogramming, they must be included in this document. Therefore, in order to avoid claiming undue credit for the linguistic developments, I have described language design decisions using first person plural ('we").

However, converting these design decisions into an implementation of HyperPascal was solely my work.

Abstract

This thesis describes an investigation into the integration of hyper-techniques with visual programming languages to support a multi-dimensional, minimally syntactic program representation.

Programming involves two phases: first, forming a mental model of the problem solution; secondly, mapping the mental model onto a physical representation. The mental model is complex, syntax-free and multi-dimensional; in textual programming languages, the physical representation is complex, syntax-rich and single-dimensional. Performing the mapping is painstaking work which has more to do with easing compilation than with representing data manipulations.

It is believed that a physical representation which better matches the programmer's mental model will significantly reduce the difficulty of generating programs.

Modern computer systems combine powerful processors, and large memories with high-resolution graphics and powerful graphic input mechanisms. This ideally fits them for supporting the building and interpretation of complex multi-dimensional structures with minimal syntax.

The Hyperprogramming paradigm exploits this capability. A hyperprogramming language uses different visual representations for different program dimensions - for example different visual vocabularies are appropriate for algorithms and subroutine nesting. Each *view* is carefully chosen to overlap the others minimally, and where overlap is essential, hyperlinks between views are provided to allow easy navigation between them, and to allow automatic updating of shared information.

HyperPascal was developed using this philosophy, as a testbed for it. In creating a program, a HyperPascal programmer edits information in three separate views:

> the *action window* view, in which subroutines are each represented using a visual language based on structure diagrams
>
> the *scope window* view, in which declarations are stored in a nested structure corresponding to conventional subroutine nesting
>
> the *forms window*, in which the appearance of I/O can be designed using a WYSIWYG editor, free of the distractions of data processing specifications.

A protoype of HyperPascal has been implemented, and a number of programs developed using it.

Acknowledgements

I would like to thank Professor Mark Apperley and Paul Lyons for their suggestions and encouragement throughout the period of this research. In particular, I would like to thank Paul Lyons for the extensive proof-reading undertaken, and guidance given in the production of this thesis.

Thanks also go to my friends; those at Massey, and those outside, for supporting me in my time at Massey and providing a welcome distraction from research when needed. Particular thanks must go to Tracey Murrell for understanding the occasional intrusion of work in my private life, and seeing me through the high and low points of my post-graduate study.

John Grundy, John Hosking, and the Department of Computer Science at Auckland University are thanked for allowing the use of the MViews framework.

The financial assistance of Massey University is also gratefully acknowledged.

My biggest thank you goes to my family; my sister Debbie, and especially my parents Bonnie, and Robert for the support and love they have always provided.

This thesis is dedicated to my family.

Contents

Chapter 1
Representing Programs

There have been many developments in human-computer interaction in the forty-four years since ENIAC was first unveiled. Plugs and switches have been replaced as input media by punched cards. Early direct control computers have been successively replaced by batch-mode mainframes, interactive single-user and multi-user mainframes, stand-alone micro's and networked systems. Output devices have changed from memory-mapped rows of lights through line printers to high resolution colour screens and laser printers. One development in particular has influenced the style of modern interfaces more than all the others. This is the direct manipulation graphical user interface, which combines mouse input, high resolution graphics monitor and multiple-window data presentation.

Most applications produced today make use extensive use of graphical user interfaces and direct manipulation. Many applications also allow users to input information visually, or produce graphical output. This trend towards graphical user interfaces and direct manipulation has, however, progressed more slowly in some areas than in others. An area exhibiting this slow progression is that of computer programming.

In the early stages of computer development, programmers were the only direct users of computers. As computers became more widespread and easier to use, another class of user, the end user, started to appear. These relatively naive users became the driving force behind human-computer interaction, and new interaction styles were developed to make applications more "user-friendly". Interaction styles in the programming process were largely ignored at this stage because programmers were considered experts at using computers, and therefore did not need them to be "user-friendly". Development in the programming process instead concentrated on increasing the power of high level languages, in order to increase productivity and satisfy the user demand.

Of course, this concentration was not total. Early work on syntax-directed editors, for example GED (Moretti and Lyons, 1986), has been incorporated into present program development systems. Most such systems perform enough parsing-on-the-fly to provide comparatively simple aids such as pretty-printing, detection of malformed expressions, and so on. However, the underlying nature of the languages they deal with is unchanged. The languages still use a single-dimensional stream of text to represent complex algorithms and data-structures. Perhaps this should not surprise us. Linearity pervades the computing milieu: we store data in sequentially addressed memory; we squeeze data through the ALU to process it. Little wonder, then, that our compilers are sequence-oriented, and that the languages they deal with are too.

However, it need no be so. We don't always design sequentially - on the contrary, in many fields, it is common practice to initiate a design with informal two-dimensional diagrams and it would be attractive to be able to capture designs in this form. Hitherto, the undoubtedly greater effort of capturing a visual programming language than a conventional textual language has mitigated against their use. Now however, high-resolution graphics, processor power and memory are now available cheaply enough to render two-dimensional visual programming languages viable.

What is so attractive about a visual language that we are willing to expend this extra effort? There are several benefits:

- Relationships are two-dimensional
  Relationships between components of a complex systems are rarely simple enough to be expressed in a simple linear fashion; more often we find that responsibilities, communication, and division of tasks require a network if they are to be properly described.

- Diagrams can use mnemonic shapes
  At least in systems with simple vocabularies, the shapes of diagrammatic components can be imbued with mnemonic significance - arrows to represent direction of flow are a ubiquitous example of this - and the significance is assimilated subconsciously.

- Partitioning is natural in diagrams
  A two dimensional surface gives a designer more freedom to partition a complex system, and to use physical proximity or separation to indicate similarity or difference between its components.

- Vocabularies can be easily distinguished
  Most complex systems have to be specified in more than one domain - programs, for example, involve algorithms and data structures. It is possible to design a visual language to emphasise the similarities and differences between the different domains. Thus, by associating a distinctive window-type with the current domain, for example, the user can be subliminally cued to employ the correct vocabulary for the current part of the design.

These benefits have been incorporated into a number of Visual Programming Languages (Shu, 1986), some of which will be examined in more detail later. For now, let it suffice that most of these languages are ways of visualising *programs*. This is an inherently flawed approach to visual programming, as it works backwards from a naturally sequential representation to a two- (or more) dimensional representation. The present work has proceeded from the basis that the object of the visualisation should be, not the program, but the programmer's mental model of the solution to a problem.

The idea of a programmer's mental model merits a little further explanation. We may describe a program as the eventual physical representation of a mental conception comprising a group of separate, but related, specifications of aspects of the solution. One such specification would be the data-processing operations which the program will have to perform. Another would be the appearance of the information input and output by the program. Clearly these two are linked - processing of information cannot begin until it has been input. However, they are also largely independent - formatting I/O is at most loosely linked to the program's data processing activities. Mapping them onto a sequential representation must at least compromise their structure, if not conceal it altogether.

1.1. Project Aim

This work aims to test the validity of this idea of using a multi-dimensional physical representation to represent the programmer's mental model. It combines techniques from the Visual Programming Languages programming paradigm and the hypermedia navigation support systems. The result has been the design of a *hyperprogramming* language HyperPascal, and the implementation of a prototype integrated program development environment for HyperPascal.

Note that there has been no attempt to introduce new programming functionality with HyperPascal, and that even the diagrams are not particularly revolutionary. The originality of the work is associated with the multi-dimensionality of the representation, and the use of hyper-techniques for navigating through the multi-dimensional space.

1.2. Statement of Thesis

We can summarise the above arguments thus:

In writing a computer program, a programmer devotes much effort to developing a mapping from a complex multi-dimensional *mental* model of the problem solution onto a *physical* representation which is sufficiently simple and syntactically rigorous for a computer to interpret. It is possible to use modern interface technology and processor speeds to support a closer match between these two representations than is available in conventional programming languages, both textual and visual. An appropriate representation would allow:

- a multi-dimensional representation of programs.
- multiple views with minimal interaction (storage of identical information in more than one view).
- where interaction is unavoidable, editing of the shared information *via* a single mechanism, identical in the different views.
- easy transitions between views.
- automatic updating of information shared between different views.
- subliminal clues to the nature of the current view.
- maximum support for navigation within the representation.

1.3. Project History

In this description, the progress of the research has been split into separate phases. In the research proper, these phases were not strictly separate, but are described this way to better show an overview of the research.

Since the aim of the project is use a visual language to provide a representation of a program that is closer to the programmer's mental model, previous research was reviewed from the areas of visual programming, human-computer interaction, software engineering, and software comprehension. A number of visual programming languages were also compared. This review identified several areas where visual programming languages were deficient in the representation of a programmer's mental model.

The field of hypermedia was identified as representing documents in a multi-dimensional manner, and techniques from the field were adapted for use in representing computer programs (a program represented multi-dimensionally is termed a *hyperprogram*).

A general purpose visual programming language, HyperPascal (Hyper - *hyper*program, Pascal - based on Pascal semantics), that used a multi-dimensional representation, was designed as a "testbed" language to test the validity of hyperprogramming ideas. A number of different views onto the program structure were identified, and first-draft notations developed for each view. During this phase the notation was most fluid: various alternative formulations were experimented with, and a consistent notation finally emerged.

To support both novice and expert programmer interaction, a number of different interaction techniques were incorporated into the design of HyperPascal, and the programming components modified to better utilise these techniques, while also ensuring that they remained consistent.

A subsidiary goal was to make HyperPascal simple for novices to learn. To facilitate this, the syntactic load was reduced by reducing the number of different programming components. This reduction in the number of component types was performed by generalising similar components into a single, specialisable, consistent component. The generalisation of the programming components also allowed a compaction of the notation, and provided generalised components that are powerful enough for use by expert programmers.

A prototype of the language was developed so that we could test the multi-dimensional representation of, and navigation within programs in HyperPascal. This implementation has concentrated mainly on the major components of the language, and their ability to specify an executable program. The prototype language enables simple hyperprograms to be translated to Pascal source code for later compilation and execution, and a number of hyperprograms have been developed and executed using this prototype.