Visual Craft: A Visual Integrated Development Environment

Ahmed F. Hegazi, Ahmed M. Metwally, Lamia M. Degady, Wael M. Abu El Saadat, Amr El-Kadi, and Sherif El-Kassas

Department of Computer Science
The American University in Cairo
113 Kasr el Aini Street, P.O. Box 2511
11511 Cairo, Egypt
elkadi@acs.auc.eun.eg

Abstract

Visual Craft is an ongoing project to develop an integrated software development environment which uses visual languages and notations to support Object-Oriented development. Visual notations are used to provide an intuitive notation for both system design and implementation. The Visual programming component is based on a well defined graph grammar that enables the systematic construction of such environments, as well as syntax directed translation and code generation. The programming environment is modeled around interactive, derivation based, syntax directed editing. This paper outlines the basic components of Visual Craft and gives a more detailed account of the visual programming aspects of this environment.

1. Introduction

Visual Programming is a relatively new paradigm that aims at exploiting the, commonly accepted, observation that the human mind is better optimized for multidimensional data in software development. As Shu asserts "it aims at exploiting our non-verbal capabilities"[1]. The fundamental elements of this approach are visual adornments, which are employed to represent programming constructs. It is asserted that such adornments, if well designed, can convey more information than the corresponding textual representation. This arises from the fact that one of the two hemispheres constituting the human brain is dedicated to the perception of non-verbal visual patterns [1].

Early attempts to create visual environments, whether general purpose or domain specific did not succeed in grasping the attention of the software industry [2,3]. However, as Shu has noted, the continuos decrease in hardware prices in general, and in graphics related hardware in particular, acted as a catalyst, for research in Visual environments [1]. In fact, statistics and analysis predict that $3.79 Billion will be allocated annually for the research in this field by 1999[4].

Visual Craft is an integrated development environment that uses visual programming and visualization to provide an interactive framework for system development. Figure 1 gives a classification of visual systems showing Visual Craft's position in this classification.

The remainder of this paper outlines the basic components of Visual Craft and provides a more detailed account of the visual programming aspects of this environment.

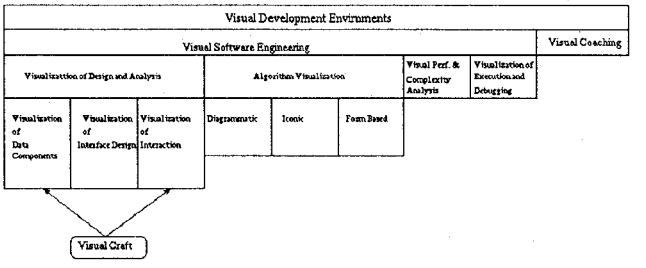

Figure 1. Classification of Visual Environments

2. Overview of Visual Craft

Visual Craft is a Interactive Visual Development Environment that consists of a Visual Programming Language and a supporting design tool (see Figure 2). It is intended for the professional software engineer who

0-8186-7852-6/97 $10.00

would use it to develop any Object Oriented Application without restrictions on its type. The design tool will aid the software engineer to formulate his design of the application in need. Upon the software engineer request, the system should generate an intermediate representation for the design and implementation. Finally the system's code generator should generate the equivalent Object oriented code of the desired object oriented language, along with the suitable Make file for the project, and an executable release of the application. Although, there is an inherent sequence suggested by the four sub-systems, yet, the environment allow an iterative development process. In this manner you can produce several releases before the final one, as you can start with a small subset of your specification then you keep on augmenting it.

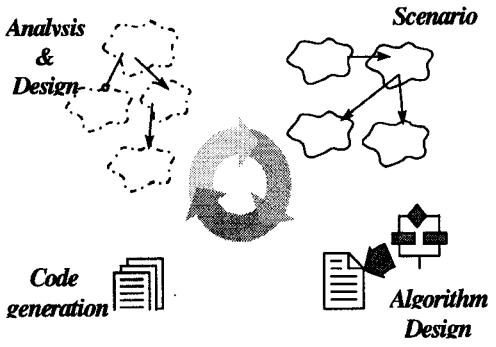

Figure 2. VisualCraft's Four Main Subsystems

Analysis and Design

In this phase the user identifies the different classes of the system along with there relations. The user should also specify the interface of these classes. We have chosen to adopt the Booch method for analysis and design as it captures both views of the design: the physical and the logical views.

Scenario modeling

This sub-system is used to define the execution flow of the application via an enhanced object model. This tool facilitates the imagination and development of different possible scenarios which is the natural way for the mind to solve a given problem.

Visual Programming

For each method in each class, the user will need to specify his algorithm using visual adornments. We adopt the term 'algorithms' because the user is abstracted from implementation details and syntax issues which allows him to concentrate on the logic of the method.

Code Generation

At this stage, the classes, objects, and algorithms which has been already defined by the system's user, will be translated to the desired object oriented language. The constraints specified through the classes' relationships and objects' interactions will be considered during the translation process.

3. The Visual Programming Component

Considering the four sub-systems constituting ***VisualCraft***; the most challenging one, indeed, was the visual programming sub-system. This was mainly due to the fact that visual programming notation had to be general purpose and not domain specific, and also due to a number of important design and implementation decisions.

In the visual programming sub-system we had to take several major design decisions. First, the most important and awaited question: how will the programming constructs be represented visually? Excluding virtual reality and scientific fiction movies stuff!, we were left with two choices: icons or diagrams. Two factors had to be put into consideration. First, the limited screen space and the nesting of program blocks. Although, icons consume much less space than diagrams, yet, a problem would immediately appear, that of nested blocks. A proposed solution is to divide the user workspace, into several independent smaller workspaces, and upon the user request (e.g.: clicking on an icon representing an iteration); another workspace would be displayed, which contain the code within that iteration construct, also represented in the form of icons. However, this solution created a serious problem with the visualization of the visual code!; well written code usually contain up to three, and in some cases four, levels of nesting. This means that for the user to reach from the outer block to the inner most block, he has to pass by several workspaces: a tiresome job that do not even exist in textual languages. Moreover, due to limited screen space, the user of our visual environment will not be able to "see" his nested code in one screen, as it was expected that each independent workspace will occupy almost half the screen. That is why we have decided to use diagrams; they can be easily nested within one workspace through applying stretching techniques. As for the problem of screen space, we provided the user with a scaling facility which maximizes the visual code per screen.

- ***Linear Vs non linear.***

The second major design decisions was whether to parse the visual adornments in a linear or a non-linear

fashion. By non-linear parsing, we mean that the positing of diagrams/icons in the user workspace would in itself entitle a specific meaning. Thus if the user puts an icon/diagram to the left of another icon/diagram, this would mean differently than putting the same icon/diagram to the right of the other. Non-linear parsing has been adopted in several domain-specific visual programming environments, basically because the nature of the problem required such mechanism. On the other hand, linear parsing provides a simpler approach which is adopted by most textual programming languages (e.g.: C, C++, Pascal, ..etc.). The only positing principle involved is that when construct X comes before construct Y, X should be parsed before Y. This is achieved implicitly by the parser working on the tokens supplied by the scanner. Simply, because the later scans the code file from beginning to end. The fact that we were designing an environment that could be used in implementing complex systems compelled us to choose linear parsing. First, by adopting linear parsing, there will be a one-to-one correspondence between our visual language and all textual languages. Second, we see no real advantage associated with adopting non-linear parsing for the algorithmic notation developed for visual craft. The only disadvantage incurred by adopting linear parsing is that the free-hand editing of visual constructs is set aside. We mean by free hand that the user will select and place visual adornments in the work space, without the system being able to deduce any semantic meaning form the relative position to each other, e.g. construct X is above construct Y. Thus the system will be able to parse the constructs only when the user established connections between them. The fact that programmers do not write blocks of statements and re-arrange them to produce a meaningful unit encouraged us to sacrifice free hand editing.

- ***The parsing is done interactively.***

As we where designing our system, it was necessary to decide whether the programming subsystem will be passive, till compilation time, like most text based compilers or will it interact with the user and show him his errors. We decided that the system should be interactive, so that the user will end up with the minimal set of errors: his own logical errors. However, we chose only to warn him using signs and errors messages without forcing him to correct errors immediately. This is mainly because a typical programmer would write a piece of code knowing that it is incorrect and then correct it later. For example, a programmer decides to use a new variable and includes it in an expression prior to declaring it. For this reason, we thought of only reminding the user that there is something wrong in his code but not to force him to correct it on the spot (see Figure 3, unknown identifier). He can save and edit his code even if it is erroneous, however, code generation will not take place except if the code is syntactically and semantically correct.

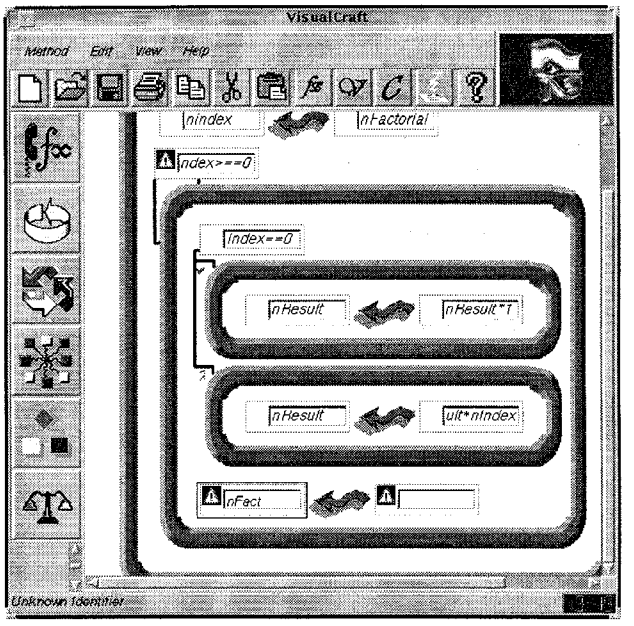

Figure 3. Unknown identifier

- ***Grammar***

The visual grammar was first written in a BNF-like form. Yet, it was found out that the BNF--like notation is not practical for the system's needs as the resolution of the BNF rules creates an overhead on the interactive parser. We concluded that the most suitable notation for the system is the one which eliminates the need for multiple rules to represent a recursive structure, like a statement-block, and also eliminates the inter-symbol separators, like ands & ors. That is why, we developed a notation which eliminates these representations from its syntax and exports them to its semantics. For example, instead of writing a recursive construct in 2 rules, it can simply be written in a single rule, by just specifying the type of the rule as being recursive. This does not just minimize the number of rules in the grammar, but it also minimizes the number of non-terminals in the parsing tree. In the same manner, instead of having a certain separator in the rule specifying that two symbols are orred, it can be simply found from the type of the rule, whether its an And or an Or rule. This rule typing mechanism was an advantage to our system as it eliminated unneeded steps in the rule resolution, for example if we had the symbols' separators we would need to build an extra mechanism to look ahead and deal with

extra symbols in order to recognize whether two symbols in a rule are orred or anded.

This type of grammar has some limitations. For example, a rule's left hand side must be unique, but you can overcome this limitation by compacting all the rules in one and then giving this rule the orred property. Also, a rule cannot contain both anded and orred symbols. But, these limitations are actually advantageous to interactive systems, as they minimize the time taken to lookup for a left-hand side's resolution, since it is known that each non-terminal has a single rule only. They also, minimize the resolution mechanism as the right-hand side symbols are either all anded and thus must all exist, or are all orred and thus only one of them is sufficient for the resolution.

- ***All visual or a hybrid of visual and text.***

A major question which poses itself while designing a visual system is how far we intend to use visual adornments. In other terms, will the user click buttons for every action desired, even if it is a simple arithmetic expression, like x + y / 10, or are we going to allow the user to edit text. We decided to allow typing, only, at the very detailed level, like expressions and identifiers, while the visual adornments are used for more conceptual needs.

- ***Integrating two parsers (role of the expression parser).***

Due to the fact that we incorporated both textual and graphical information into the programming subsystem we had to parse both graphs and texts. Therefor, we had to have two parsers instead of one. These parsers coordinate to insure that all errors, whether in the graph or in the text, are detected, with the exception of the logical errors.

Before explaining the role of both parsers we should name them to avoid confusion. The parser responsible for parsing textual information is called the *expression parser*, while the graphical one is called the *visual interactive parser* or for short the *visual Parser*. Whenever a textual expression is written in one of the visual constructs (e.g., a condition in a iterative construct), it will be parsed by the expression parser which will catch any syntax error using a grammar made for expressions. Moreover, the expression parser, uses the symbol tables to detect semantic errors, such as, unknown identifiers, inappropriate parameters, etc.. In case the parsing of the textual expression was done successfully, syntactically and semantically, the expression is passed to the visual parser who would consider this pack as a terminal token. Next, the visual parser will execute any semantic routine associated with the grammatical rule of the terminal. In this phase any errors found will be registered inside the system and displayed to the user in the form of a warning sign. For example, an assignment statement is composed of an identifier, equality operator, and an expression. If the user changes the identifier, the expression parser will recheck it. In case the new identifier is valid, syntactically and semantically; the visual parser will execute the associated semantic routine; checking for type mismatch between the identifier and the expression.

The visual parser also checks for syntax errors. This occurs in the case of addition of a new construct or an editing operation on one or more constructs. The visual parser always refer to the grammar in the above cases to ensure that it follows the production rule. For example, if the currently selected object on the screen was an *expression* of an iterative construct and then the user tried to add an *if* statement inside that *expression*, the visual parser will check the grammar and will find that an *expression* cannot contain an *expression*. Therefor, an error message will appear. As a conclusion, the cooperation between the two parsers ensures the syntactic and semantic correctness of the method being designed, as the code generation will not take place unless the user removes all the warning signs.

- ***The visual parser operates on a tree and not on a stack.***

The visual programming sub-system depends mainly on the interaction between its interactive visual parser and its expression parser. As the parsing is being done interactively, the parser's information had to be kept in a dynamic container. The idea of having a stack working as the information's container was rejected from the beginning since it is costly to edit or delete information from a stack and next re-parse it. It was found that all linear data management structures are so costly, as well being inefficient for the purpose of this sub-system. Thus, we diverted our interest towards n-dimensional data containers as they are by nature inexpensive in their insertion and deletion mechanisms. At the beginning, a complete graph was considered, yet we found it unsuitable, as many of its links would be redundant. When the programming sub-system's grammar became fully developed, we realized that the language's constructs are more of containers to each other; possessing a parent-child relationship. For example, a *block* of code consists of an *if* statement, followed by a *while* statement. The most suitable container for this type of relation was found to be the tree structure. Nevertheless, the n-dimensional tree did not provide the maximum efficiency when handling leaf or terminal nodes. Simply because, these

nodes are the expressions which need to be accessed by the expression parser. Thus, a faster access mechanism is needed, to achieve re-parsing as quickly as could be. Therefore, we linked all terminal nodes together using a random access list to insure the efficiency of both the interactive parser and the expression parser.

- ***Need for a separate locator for visual adornments, and some mean of screen memory mapping.***

As there was a great deal of user dependent activities in the system, and because it is a visual environment, the parsing information and their representation on the screen where strongly coupled.

As a rule, parsers should not handle display and I/O issues, yet there exists strong coupling between the parsing information and their representation on the screen due to the nature of the system. This arose the need for another entity in the system to maintain the relation between the displayed diagrams and their corresponding information. This entity : *Locator*, locates nodes in the parsing tree corresponding to the selected diagrams on the screen. Through, its algorithms and the parsing tree's information, the *locator* is capable of keeping the correspondence between the display and the parsing information without breaking formal object oriented design rules. Consequently, maintaining separation of concerns in the system. In other words, the *locator*'s job combined with that of the GUI is to emulate the job of a lexer which reads a stream of graphical diagrams and feeds tokens to the interactive parser.

- ***IR generator and Code generator***

The final output of the visual programming system was an important design decision. On one hand, we had the alternative of producing binary representation of the user visual code. A clear disadvantage is the lack of portability, due to the differences in machine's architecture. Hence, we decided to generate an intermediate representation (IR) which is platform independent. Thus, the system's users can work on different platforms, and yet can flexibly integrate their work. Creating our own representation of the visual code aided us in two aspects: first, the security of the user's visual code is guaranteed, throughout the development process. Second, it allowed us to make the IR compatible with the data representation in memory, and therefore, speeding up the process of saving, loading and modifying the visual code.

The intermediate representation can be mapped, through the system's code generator, to any object oriented programming language. We have decided to supply an ANSI C++ code generator. Our choice was based on the fact that C++ has almost become an industry standard. Nevertheless, other code generators can be easily plugged in our system, to generate other languages: Object Pascal, Ada , or Smalltalk. Indeed, this approach gave us the opportunity to make use of existing compilers, and the powerful optimization techniques provided by some of them. We believe that this is one of the most powerful features of our visual language, as the user's visual code can be mapped to the programming language that has been previously used in the development of other parts of the target application.

4. Summary and Future Work

Visual Craft is an ongoing project to develop an integrated software development environment which uses visual languages and notations to support Object-oriented development. This paper outlines the basic components of Visual Craft and gives a more detailed account of the visual programming aspects of this environment. It also explains many of the design and implementation decisions that were addressed while developing visual craft. The current systems supports visual programming and enables users to generate C++ code. Future plans include the full integration of the Booch notation, the addition of extra code generation modules to enable the generation of Java code, and the development of a powerful visual class library.

References

[1] Shu, N.C. *Visual Programming*. Van Nostrand Reinhold Company Inc. 1988, pp. 1-20.

[2] Glinert, E.P., and S.L. Tanimoto. "PICT: An Interactive Graphical Programming Environment." *Visual Programming Environments : Paradigms and Systems*. Ed. Ephraim P. Glinert. California: IEEE Computer Society Press, 1990.

[3] Kimura, T.D., and J.W. Choi . "Show and Tell: A Visual Programming Language." *Visual Programming Environments : Paradigms and Systems*. Ed. Ephraim P. Glinert. California: IEEE Computer Society Press, 1990.

[4] Snell, Monica 'Analysts predict $3.79 Billion market for visual development tools by 1999'. *IEEE Computer*. Vol. 28 no 3 March 95, pp. 8-9.