LIVE - Integrating Visual and Textual Programming Paradigms -

Keiji Kojima, Yoshiki Matsuda, Seiji Futatsugi
Central Research Lab., Hitachi Ltd.

ABSTRACT
Recently, there has been a great interest in visual programming languages. Visual programming languages are easy and friendly, however they are inferior to conventional textual languages in their preciseness and generality. This paper demonstrates that visual programming languages and textual programming languages are not rivals but complement each other. A three-dimensional animation-oriented programming language called LIVE is presented, which is an attempt to integrate the visual programming paradigm and the textual programming paradigm. Visual objects are interactively manipulated by both a visual representation (pick-by-position interface) and a textual representation (pick-by-name interface). The system informs the user in a text form how his visual operation is interpreted. A selected interpretation unsuitable to the users can be interactively resolved using a pick-by-name interface. LIVE introduces a programming-by-multiple-examples paradigm based on a visual guard concept, which is analogous to the concept of logic programming languages. This paradigm enables one to make a visual program in way which is simple and easy to understand.

1. Introduction
With the rapid progress of VLSI technology, the cost effectiveness of computer hardware has improved dramatically. On the other hand, computer software productivity remains at a low level, and a shortage of programmers has become a serious problem. This crisis, however, is not the first of its kind. At one time a shortage of telephone operators presented a big problem in the rapid increase of telephone networks. A shortage of car drivers at the start of the century was another example. Both of these problems were solved in the same way. The systems (telephones and cars) were modified in a way that the users themselves could operate them.

Unfortunately, in the domain of computer software, it is difficult for end-users to play a programmer's role. A main obstacle exists in conventional programming languages. They require sophisticated symbol manipulation skills which are difficult to learn [Glinert 84]. In order to realize end-user programming, it is important to design a suitable programming language. Recently, there has been a great interest in visual programming languages as a promising approach [Chang 86] [Myers 88]. Without a doubt visual programming languages are easy and friendly [Glinert 84] [Fine 84] [Tanizoto 86] [Hirakawa 87], however they are inferior to conventional textual languages in some respects; in particular, preciseness and generality [Shu 86].

In debates on programming methodology, visual programming languages and textual programming languages are often considered to be rivals. However, “visual or textual” need not be an exclusive choice. The objective of this paper is to demonstrate that visual programming languages and textual programming languages are not opponents but can complement each other. We present a programming system called LIVE (Language for Intelligent and Visual Environment, pronounced [laivl]), which is an attempt to integrate a visual programming language and a textual programming language. Its design principle is discussed in Section 2, and in Section 3 an application scenario is demonstrated. In Section 4 an implementation is described and Section 5 presents the conclusion.

2. LIVE Design Principle
In this section, we point out problems caused by using a visual or a textual programming language alone, using examples from LIVE. In the subsequent sections, all types of visual programming languages are not discussed. Instead, we concentrate on a class of iconic languages which are characterized as follows:

(1) Data objects are represented by graphical icons.
(2) A data object to be processed is selected by using a pick-by-name interface.

For example, when a user wishes to compute f(a, b), the user first picks data icons representing a and b, then selects function name f from a menu. This characterization is utilized in number of direct-manipulation iconic systems [Shneiderman 83]. Other important classes of visual programming languages, such as flowchart-type languages, however, are out of our scope.

LIVE in part belongs to the class defined above. The way the system appears is shown in Figure 1. Programming is performed on 3-D scenes. Visual data objects such as numbers, arrays,
humans, and so on are placed on the scene. Selected objects are highlighted for easier identification. A function is specified by its name, and can be inputted from a menu or the keyboard. The effect of function applications is represented by using animation. For example, if a user picks a human and applies a function "walk", the human walks in the scene. The recorder-like operation panel on the scene frame controls program execution (step-wise execution, continuous execution, and so on). LIVE is based on programming by example paradigm [Halbert 84], which we will discuss in more detail later.

2.1 Operating Atomic Data Objects

It is not a difficult task to specify argument data objects using a textual programming language. A programmer only needs to type their names correctly. It seems to be easier to do the same job using a visual language. In fact, a programmer can specify a data objects with one touch even if he does not know their names. In visual languages, however, there is a problem that the order of selections is not clear. For example, as shown in figure 2, a user has selected numerical objects m, n, and m in that order. If the user forgets what order he has picked them, the order information cannot be retrieved from the visual representation. An auxiliary information system is necessary to give that precise information. In LIVE, the textual information, which corresponds to a visual operation, is displayed as a script (shown in the right upper corner on the scene). A script is strictly compatible with a visual operation. For example, picking a data object by a mouse and inputting its name from keyboard has exactly the same meaning. The first operation is called pick by position, and the second one is called pick by name. Experienced users may prefer the pick by name interface which is also efficient in the case where a 3-D scene is crowded with many objects.

2.2 Operating Structured Data Objects

It is a sophisticated task to manipulate structured data objects such as arrays, lists, trees, graphs, and so on using a textual language. A programmer must keep their graphical model in mind. It is the programmer's responsibility to translate the model to a one-dimensional textual representation. An inconsistency between the model and the textual representation consequently becomes a bug.

On the other hand, visual languages are powerful in operating structured data objects. In visual languages, structured data objects are displayed as their graphical images. A programmer can manipulate these images directly. Therefore, no mapping effort is needed. The success of spreadsheet software packages is proof of this merit.

Unfortunately, there is an ambiguity in visual languages, as illustrated in Figure 3. The figure shows the scene status after a user has selected the highlighted element in the linear array a. The upper arrow represents an index pointer i. Multiple interpretations of this scene are possible. For example, the user has picked the third element of a, the user has picked the indexed element of a, the user has picked the central element of a, and so on. In textual languages, the ambiguity can easily be avoided. For instance, textual representations a[2], a[i], and a[(n−1)/2] clearly distinguish the above cases (n being the length of a).

There are two ways to resolve this ambiguity.

1. The system presents its interpretation in a textual form when a structured data object is operated. If the interpretation by the system is not correct, the user edits the text appropriately.

2. The user operates a structured data object in some predefined gesture. For example, if we have a visual sign syntax rule "to click the index icon twice means to pick the indexed element". In this way a[2] and a[i] can be easily distinguished.

Figure 2. Visual Operation Ambiguity(Atomic Data Objects).

Figure 3. Visual Operation Ambiguity (Structured Data Objects).

Figure 4. Extended Selection

LIVE adopts the former approach. As shown in Figure 3, the system generates the scripts "a[i]", which is the default interpretation. If incorrect, the script is directly modified by a user. Currently we do not employ the second approach since it is feared that the sign rule system may become too complex for users. Of course, some visual gestures are natural and useful.
highlighted subarray by dragging the mouse from the first element to the indexed one. It should be noted that displaying the corresponding script \( [0 - i] \) is a good way of confirming the formal meaning of the visual gesture.

For visual languages, it is important to construct widely acceptable visual sign rules. Consequently, to standardize visual representations and operating gestures on these requires a significant amount of work.

2.3 Programming By Example

Many visual programming languages are based on programming by example (PBE). In this method a procedure is obtained from the history of direct manipulations on example data objects. Typically the user inputs manipulations of example data objects representative of his application. The method is characterized as follows:

**PBE scheme**

Let \( \Pi_0, \ldots, \Pi_m \) be an operation sequence that a programmer has made for input data objects \( a_0, a_1, \ldots, a_n \). Each \( \Pi_i \) is considered to be an n to n mapping procedure. Then, \( \Pi_0 \circ \Pi_1 \circ \ldots \circ \Pi_m \) is obtained by example \( a_0, a_1, \ldots, a_n \). The notation \( \Pi_0 \circ \Pi_1 \circ \ldots \circ \Pi_m \) denotes a compound procedure of \( \Pi_0, \Pi_1, \ldots, \Pi_m \), and \( a_0, a_1, \ldots, a_n \) represent formal parameters. A set of input data objects \( a_0, a_1, \ldots, a_n \) is called an example.

Note that it is not such a straightforward task to implement a PBE system as we described in the previous subsection, the above \( \Pi_i \)'s are not uniquely obtained because of the direct manipulation ambiguity. That is, in order to obtain the correct sequence of \( \Pi_i \)'s, the ambiguity resolving mechanism described above is indispensable.

PBE is especially suitable for non-programmers since it enables them to make a step-wise confirmation of a programming process (Lieberman 88). The problem of PBE lies in its inability to represent execution flow controls such as a conditional branch. In existing PBE systems, flow controls are specified by explicit control commands. For example, suppose that a programmer wishes to execute a procedure \( q \) after \( p_i \) only if a condition \( b \) holds. Firstly he will give a branch command. Secondly, the system will ask him the branch condition and the branch target procedure name using a dialog box. After giving that information, the system recognizes \( p_i \) is "if \( b \) then \( q \)". This method is simple, but spoils the advantage of PBE. A programmer must keep exceptional cases in mind, which are not displayed in the current example.

The other way to represent execution flow controls in PBE is to introduce an execution scheme based on a success-failure model. In order to explain this method, we will first discuss the extension of PBE in the case of multiple examples. Let \( q_0(0), \ldots, q_0(N_k) \) be an operation sequence that a programmer has made for \( i \)-th example \( a_0(0), a_1(0), \ldots, a_n(0) \). Then \( q_0(0) \circ \ldots \circ q_0(N_k) \) is integrated to a single procedure \( P \). This scheme has a strong resemblance to a logic programming language such as Prolog (Clocksin 81). When \( q_0(0) \) is regarded as a predicate and \( q_0(0) \circ \ldots \circ q_0(N_k) \) is regarded as a single Horn clause, the scheme can be rewritten in the following form:

\[
P :: q_0(0), q_1(0), \ldots, q_{N_k}(0).
\]

This similarity implies the following LIVE semantics based on the success-failure model with multiple examples.

**LIVE execution semantics**

(1) Every primitive visual operation ends in either success or failure.

(2) \( q_0(0) \circ \ldots \circ q_0(N_k) \) from the \( i \)-th example and \( q_1(0) \circ \ldots \circ q_{N_k}(0) \) from the \( j \)-th example are called alternatives of each.

(3) An execution of \( q_0(0) \circ \ldots \circ q_0(N_k) \) succeeds if all \( q_0(k) (k = 0, \ldots, m-1) \) end in success. Otherwise, the execution fails.

(4) An execution of the integrated procedure \( P(0, b_1, \ldots, b_k) \) is defined as:

(i) Select an alternative, say \( q_0^0(0) \circ \ldots \circ q_{m-1}^0(0) \).

(ii) Execute \( q_0^0(0) \circ \ldots \circ q_{m-1}^0(0) \) (\( b_1, \ldots, b_k \)).

(iii) If this execution succeeds, let \( P(0, b_1, \ldots, b_k) = q_0^0(0) \circ \ldots \circ q_{m-1}^0(0) \), and the execution of \( P \) ends in success.

(iv) Otherwise, try other alternatives.

(v) If all trials fail, \( P \) fails in failure.

Figure 5 shows a LIVE procedure which is a visual implementation of the well known partition process used in quicksort (Knuth 73). In LIVE, a scene is described by a single example. Therefore, the script of the scene represents an alternative. The part procedure consists of four alternative scenes. The procedure has two arguments, the pivot \( M \) and the array \( A \). The first scene (Figure 5(a)) is described by example values \( M = 5 \) and \( A = [6, 8, 2, 4] \). The second scene (Figure 5(b)) handles the case that \( A \) is an array. The third scene (Figure 5(c)) describes the case that the pivot is less than any element of \( A \). Similarly, the fourth scene (Figure 5(d)) shows the case that the pivot is greater than any element of \( A \). Figure 5 also illustrates the following LIVE language syntax:

**LIVE syntax**

\[ <\text{procedure}> ::= [<\text{alternative scene}>]. \]

\[ <\text{alternative scene}> ::= <\text{instruction}>. \]

\[ <\text{instruction}> ::= <\text{object-pick}> | <\text{procedure-pick}>. \]

\[ <\text{object-pick}> ::= <\text{pick-by-position}> | <\text{pick-by-name}>. \]

\[ <\text{procedure-pick}> ::= <\text{pick-by-position}> | <\text{pick-by-name}>. \]

As shown in Figure 5, readability of a program has been improved because each alternative scene can be understood independently. When the partition procedure is executed with new arguments, these four alternative scenes are successively tried until finding an alternative which ends in success.

In each scene, there are visual operations describing the characteristics of the given example. For example, in the first script of the second scene, a programmer has selected \( A \) and confirmed that it is null. Such a confirmation operation is called a visual guard. A visual guard prevents the execution from proceeding to a wrong portion. A visual guard essentially has the same meaning as a guard predicate of Guarded Horn Clauses (Ueda 85) when trials of alternatives are performed in parallel.

82
The current implementation of LIVE tries alternative scenes sequentially.

In LIVE, if all alternatives have failed for input parameters \(a_0, \ldots, a_n\), the system automatically makes a new scene with the example \(a_0, \ldots, a_n\). By describing this scene, the program can now handle new input patterns represented by \(a_0, \ldots, a_n\). This feature is called OJT or "on the job training" which is demonstrated in the following section.

3. An application scenario

In this section, we describe a short application scenario of LIVE. A procedure hanoi to solve the Hanoi-tower problem [Wirth 76] is to be made. We start by describing the simplest case where only a single saucer is to be moved. First, a new scene named hanoi is created. Secondly, example arguments \(N\) and three Hanoi towers \(a, b,\) and \(c\) are located in the scene. \(N\) represents the number of saucers to be moved. Saucers must be moved from tower \(a\) to tower \(b\) using tower \(c\) as a workspace. Figure 6(a) shows this status.

In the case of Figure 6(a), the problem is straightforward. That is, the following operations are performed:
1. Confirm the value of \(N\) is equal to 1. This is a visual guard.
2. Select \(a\) and apply the procedure pop (Figure 6(b)).
3. Select the popped saucer and \(b\), then apply the procedure push.
4. End (Figure 6(c)).

Now, we proceed to the case where more than one saucer is to be moved. First, a new example is created as shown in Figure 7(a). Secondly, the procedure hanoi is applied to this example. This application fails because the visual guard \(N \neq 1\) fails for this example. Therefore, the on the job training mechanism creates a new scene with this example (Figure 7(b)). This time, the following operations are performed:
1. Confirm the value of \(N\) is greater than 1.
2. Decrement \(N\).
3. Select \(N, a, c, b\) in this order and apply the procedure hanoi (Figure 7(c)).
4. Select \(a\) and apply pop (Figure 7(d)).
5. Select the popped saucer and \(b\), then apply push (Figure 7(e)).
6. Select \(N, c, b, a\) in this order and apply the procedure hanoi.
7. Increment \(N\).
8. End (Figure 7(f)).

Applying two hanoi alternatives according to the semantics described in the previous section, a general case can be handled (Figure 8).

4. Implementation

The current LIVE prototype has been implemented on Sun workstations as a client of NeWS window server [Sun 87]. All figures in this paper are the screen hardcopies of the prototype. The implementation language is C except the graphics control routines which are coded in PostScript [Adobe 87]. PostScript is a nice window description language, but the lack of 3-D and animation features is its weak point. Presently, we are rewriting the graphics routines using PHIGS, a 3-D graphics package.

The visual object modeler and the motion coordinator, which are not described in this paper, are under iterative design and implementation. They are interactive tools for designing 3-D
Figure 6(a). Only a single saucer to be moved from a to b.

Figure 6(b). Moving a saucer from a to b.

Figure 6(c). A saucer has been moved.

Figure 7(a). Two saucers to be moved from a to b.

Figure 7(b). An alternative scene is created (OJT).

Figure 7(c). Moving a saucer from a to c.

Figure 7(d). Popping a saucer from a.

Figure 7(e). Pushing a saucer onto b.

visual objects such as Hanoi towers. These components enhance the expressive power of LIVE.

5. Conclusion

This paper has presented a programming language LiVE, which is an attempt to integrate the best qualities of visual programming languages and textual programming languages. The bidirectional compatibility between a visual representation and a textual representation add generality and preciseness to this
visual programming language. A programming-by-multiple-examples paradigm based on a success-failure execution model makes a visual program easier to understand. A major application target of LIVE is to describe an electronic world whose primitive version is shown in Figure 1. More research on parallel execution semantics is necessary to attain this goal.

6. Acknowledgments

We gratefully acknowledge Seiichi Yoshizumi at Hitachi Central Research Laboratory for his continual support and valuable suggestions. We also would like to thank Dr. Michiaki Yasumura, Tsuneya Kurihara, and Yasusi Kanada for many stimulating discussions. Our special thanks go to Dr. Tsuneyo Chiba, to whom we owe the naming of “LIVE”.

7. References

[Ueda 86] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the concept of a guard, TR208, ICOT, 1986.