A Generic Model for Constructing Visual Programming Systems

Masahito Hirakawa, Makoto Yoshimi, Minoru Tanaka, and Tadao Ichikawa
Faculty of Engineering, Hiroshima University
Shitami, Saijo-cho, Higashi-Hiroshima 724, Japan

Abstract

The visualization schemes for visual programming are classified into two categories: One is the visualization of the flow of data/control and the other is that of spatial relationships among visual objects. In the latter case, programs are represented in terms of the change of spatial relationships.

In this paper, we propose a model for constructing visual programming systems. The model is called the Visual Programming Space (VPS). VPS is capable of supporting both of these two visualization schemes mentioned above. The Visual Programming Space Manager (VPSM), which is a system to manage programming facilities of VPS, is also proposed. Any specific programming system/environment can easily be constructed by using VPSM. An iconic programming environment, HI-VISUAL'89, is presented as an example for demonstrating the validity and effectiveness of VPSM.

1. Introduction

The development of computers with friendly user-interfaces is strongly requested so that even computer non-specialists can operate computers. One of the successful approaches to attain this is the utilization of visual information in a man-machine interaction [1], [2]. The user can specify his/her demand by simply selecting visual objects and moving them to proper positions on a two-dimensional display screen. The effectiveness of this paradigm has been demonstrated by applying it to command languages in the Macintosh-like user-interfaces.

In recent years, many researchers are investigating how visual information can be applied more effectively to programming. And eventually, programming systems which enable the user to specify programs visually on a display screen have been proposed. Examples are Pict/D [3], InterCONS [4], ThinkPad [5], and HI-VISUAL'88 which the authors have proposed before [6]. These are called visual programming systems/environments.

Here, in accordance with the enhancement of programming facilities, the development of visual programming systems becomes more and more difficult and costly. Therefore, providing tools to aid implementation of visual programming systems/environments is strongly requested by the system designer.

In this paper, we first propose the Visual Programming Space (VPS) as a generic model for constructing a variety of visual programming systems. VPS is organized by a sequence of scene frames. A scene frame represents a scene to be displayed on the screen. Actions/operations which cause transition from a scene frame to another scene frame are also managed in the scene frame.

The Visual Programming Space Manager (VPSM), which is a system to manage VPS programming facilities, is similar to the systems for developing user-interfaces called UIMS [7]. UIMS helps the programmer develop user-interface programs utilizing icons, menus, windows, and so on in man-machine interactions, but does not provide mechanisms to manage the so-called programming facilities effectively.

In contrast to this, VPSM can manage both visual user-interfaces and programming. The effectiveness of VPSM is demonstrated through the implementation of an iconic programming environment HI-VISUAL'89, which is an extension of the HI-VISUAL system the authors have developed before [6].

The definition of VPS is first presented in Chapter 2. The organization and system facilities of VPSM are described specifically in Chapter 3. Finally, in Chapter 4, HI-VISUAL'89 is introduced as an example for explaining how VPMS works in developing specific programming systems.

2. Visual Programming Space

The Visual Programming Space, VPS, is a generic model for constructing a variety of visual programming systems. Any specific visual programming system can be constructed on this model. Before defining VPS, we will classify the existing visual programming systems.

Visual programming systems are categorized into the following two types in terms of the targets to be visualized:

(1) Visualization of the flow of data/control

For the systems in this category, program development is
carried out by arranging visual objects on the display screen and specifying the flow of data/control among them. Spatial relationships among visual objects themselves have no meaning. Examples of these systems are Pict/D [3], InterCONS [4], Tinkertoys [8], and HI-VISUAL'88 [6].

(2) Visualization of spatial relationships among objects

For the systems in this category, location and size of visual objects, and spatial relationships among them have particular meaning. The flow of data/control is not visualized directly on the screen. It is specified by means of the transition of visual objects displayed on the screen. Examples of the systems are PLAY [9] and Hypertext/media systems. HI-VISUAL'89 to be presented in this paper is categorized in this type.

Thus a model for visual programming systems must be able to support both types of programming schemes. The Visual Programming Space (VPS) is defined as follows.

VPS is organized by a sequence of scene frames, as shown in Fig. 1. A scene frame represents a specific scene on the display screen. It contains several visual objects such as icons and windows which are elements to be manipulated on the display and spatial relationships among these visual objects. User's actions and/or system operations specifically assigned to it, such as pressing a key of the keyboard, dragging a visual object, and activating a function associated with the user's action are also specified in the scene frame.

These actions/operations change the state of the scene and, as a result, a new scene frame is generated. It should be mentioned here that the system's response for a specific user's action, and the timing of generating a scene frame differ depending on the visual programming system to be developed. Therefore, the development of a particular system is achieved by means of specifying such knowledge described above.

A user-defined program is represented in VPS as the combination of scene frames and several additional descriptions such as a start scene frame, an end scene frame, visual object(s) to be input, and visual object(s) to be output. Knowledge for the system is specified outside VPS. Details of the system management will be described in the next chapter.

Any of the existing visual programming systems can be constructed in VPS, as explained below:

Regarding the systems in category (1), every user's action is always specified to a particular scene frame and the results of executing functions associated with the user's actions appear in the same scene frame. The flow of data/control is visualized also in the same scene frame, and there is no need to create multiple scene frames. For the development of systems in category (2), however, a program is represented by means of multiple scene frames. Each scene frame holds a certain state of programming and the programmer can go back to any of the past states by referring to the scene frame associated with that state.

3. Visual Programming Space Manager

In this chapter, we will describe the Visual Programming Space Manager, VPSM, for managing programming facilities of VPS. VPSM provides primitive commands for the management of VPS and user interfaces. The system designer can easily construct visual programming systems by using these commands.

3.1 System Architecture

Figure 2 shows the system architecture of VPSM. VPSM is composed of an interface manager (IFM), a command interpreter (CMI), and VPS which works in VPSM as a memory unit to store scene frames. A visual programming system is constructed by specifying an interpreter (IP) and an execution manager (EM).
IFM manages two tasks: One is to display scenes associated with scene frames, menus, and dialog box windows for data input. Basically, IFM displays images which have a bitmap structure. If the system designer wants to use another display scheme, he/she adds a routine for the scheme to IFM. The other task of IFM is to transform the user's action into an internal structure that the system can accept and to notify IP of the execution. In the case that the execution of certain function modules/programs is required, EM is invoked for managing the processes of the execution of commands. The use of the return value like this is also the same to the case for other commands listed below.

GoLastScene(): boolean
The newest scene frame having the largest 'scene#' becomes the current scene frame.

GoBack(scene#): boolean
The scene frame of the 'scene#' becomes the current scene frame, and scene frames whose 'scene#' are greater than the 'scene#' are deleted. The current scene frame thus becomes the newest one.

CreateScene(): scene#
This creates a new scene frame whose contents are the same to those of the adjacent scene frame, and the scene frame becomes the newest one.

ClearScene(scene#): boolean
This deletes all visual objects in the scene frame of the 'scene#'.

SetRelation(scene#, object#, relation, object#): boolean
This defines the relationship, such as ON and CONNECT, between visual objects specified. Any relationships can be defined by the system designer.

ResetRelation(scene#, object#, relation, object#): boolean
This removes the relationship between two visual objects specified.

(iii) commands for manipulation of visual objects

CreateObject(scene#, object#, image, position, size): boolean
This creates a visual object having the 'image' and the 'size', and puts it at the 'position' in the scene frame specified by the 'scene#'.

DeleteObject(scene#, object#): boolean
This deletes the visual object specified.

MoveObject(scene#, object#, movement): boolean
This moves the visual object to the extent of the 'movement.'

ResizeObject(scene#, object#, size): boolean
This changes the size of visual object from the current size to the 'size' specified.

(iv) command for management of actions/functions

Execute(scene#, action#, semantics): boolean
This associates the user's action in the scene frame with its semantics (i.e., function which may contain VPSM commands) and executes the function. This association is registered in the scene frame and is used for program execution.

(v) commands for display control

Redraw(scene#, object#): boolean
This redisplay the visual object specified, and is used when the icon image is changed.

DisplayMenu(scene#, title, menu_items): selected_item#
This displays a pop-up menu on the display. The 'title' and the 'menu_items' represent the heading and the items listed in the menu, respectively. This
command returns selected item number.

DisplayKeyInput(scene#, title): input data
This displays a window for the input of data from the keyboard, and returns input data

DisplayScene(position, size, scene#): boolean
This displays a window for displaying the scene frame specified by the 'scene#.' The 'position' and 'size' specify the position and size of the window displayed on the screen.

Table 1 summarizes the relationship between the command and VPSM modules whose contents may be changed as a result of the command execution.

<table>
<thead>
<tr>
<th>command</th>
<th>updated modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSceneDescription</td>
<td>NONE</td>
</tr>
<tr>
<td>GetAction</td>
<td>NONE</td>
</tr>
<tr>
<td>GetCurrentSceneNumber</td>
<td>NONE</td>
</tr>
<tr>
<td>GetObjects</td>
<td>NONE</td>
</tr>
<tr>
<td>RetrieveObjects</td>
<td>NONE</td>
</tr>
<tr>
<td>GetAttributeValue</td>
<td>NONE</td>
</tr>
<tr>
<td>Go</td>
<td>IM</td>
</tr>
<tr>
<td>GoLastScene</td>
<td>IM</td>
</tr>
<tr>
<td>GoBack</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>CreateScene</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>ClearSceneDescription</td>
<td>VPS</td>
</tr>
<tr>
<td>SetRelation</td>
<td>VPS</td>
</tr>
<tr>
<td>ResetRelation</td>
<td>VPS</td>
</tr>
<tr>
<td>CreateObject</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>DeleteObject</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>MoveObject</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>ResizeObject</td>
<td>IM, VPS</td>
</tr>
<tr>
<td>Execute</td>
<td>VPS</td>
</tr>
<tr>
<td>Redraw</td>
<td>IM</td>
</tr>
<tr>
<td>DisplayMenu</td>
<td>IM</td>
</tr>
<tr>
<td>DisplayKeyInput</td>
<td>IM</td>
</tr>
<tr>
<td>DisplayScene</td>
<td>IM</td>
</tr>
</tbody>
</table>

4. HI-VISUAL’89

Development of a visual programming system is carried out by defining interpreter (IP), execution manager (EM), and user interface. Here let’s take the development of an iconic programming system HI-VISUAL’89, as an example, which is an extension of HI-VISUAL’88 the authors have developed before [6].

4.1 Icons

Like HI-VISUAL’88 developed previously [6], in HI-VISUAL’89, icons represent objects actually existing in a real world, such as papers, sales books, folders, and calculators. They have both data and functional properties just as objects in a real world do. Here it should be noted that several different functions may be assigned to a single icon. The behavior of an icon cannot be fixed uniquely but is determined at the time of programming in relation to another icon combined with it. Each icon can take an active or passive role against the other.

For example, when a pen icon and a paper icon are combined together, the pen would be interpreted as a functional tool for writing on the pen. However, the pen is treated as an item (a passive object) to be thrown away when the pen icon is combined with a trash box under the interpretation that the trash box is active in the sense of the pen being put into it. In an actual programming environment, the combination of icons is indicated by the overlapping of an icon on another.

In addition to the facilities described above, HI-VISUAL’89 allows icons to have different image icons even though they belong to the same type (for example, paper). Icon image, such as the shape and size, is decided when the icon is displayed, in accordance with the value(s) and the state of the object. For example, even if there are two ‘paper’ icons, the size of these icons may be different according to the attribute value ‘size.’ Another example is that, in a LSI CAD environment, a gate (icon) can be highlighted to show that it is in an active state. The notion of the dynamic icon image generation was first proposed by Hsia et al [10]. Dynamic icons in their systems can manage data but not functional properties of objects.

4.2 Program Specification

Figure 3 shows an interface of HI-VISUAL’89 at a particular time in the process of programming. Every object displayed on the screen, such as a sales book, a calculator, a desk, and a secretary is managed as an icon in the system.

![Figure 3 User interface of HI-VISUAL’89](image-url)

On a display, a cursor is represented in the form of a hand. Program development proceeds by moving an icon and overlapping it on another by using the hand indicated on the display screen. Let us assume, as an example, a case where we
make a program for summing up total sales, drawing a graph of the total sales, and putting the graph into a folder which is inside of a drawer.

To do this, the user first overlaps a calculator on a sales book. The system then decides the behavior of these icons by referring to a knowledge-base. A scheme for interpreting icon behavior is given in [6]. Assume that, in this case, the sale amounts indicated in the sales book are summed up according to the sales date and a paper showing the resultant data is generated. The display changes its appearance as in Fig. 4.

The user proceeds programming by overlapping a drawing kit on the paper. Then a paper showing the graph of total sales is generated. In addition, by pointing a drawer by the cursor, a folder comes out on the display. Finally, the graph paper is overlapped on the folder.

The sequence of these actions can be defined as a program by specifying a start scene, an end scene, icon(s) to be input, and icon(s) to be output. A secretary (icon) acquires the program. Once the program is registered as a job assigned to the secretary, the program is activated simply by handing over the input icon(s) to the secretary.

The display would change its appearance every time the user's action is operated. Once the display is changed, it is not possible to see the previous scene any more. The system must therefore provide facilities to present a history of user's actions. In HI-VISUAL'89, the following two methods are provided:

1) The sequence of user's actions is displayed like a film at the left side of the screen, as shown in Fig. 5. Each frame shows a spot at which an overlapping operation occurred. The frame pointed out by a triangle represents a current frame, and the scene corresponding to the frame is displayed on the screen. By properly selecting the current frame, the user can obtain any scene of the program.

2) The program structure is visualized by means of a flow graph as demonstrated in HI-VISUAL'88 [6] developed previously. The flow of actions appears as shown in Fig. 6.

4.3 Construction of HI-VISUAL'89

Figure 7 shows a part of the description for construction of HI-VISUAL'89. The bold-faced characters represent VPSM commands. The underlined functions are those programmed by the system designer himself.

This program is explained as follows: When a mouse button is pressed/released, IFM notifies IP of the invocation of the action. IP receives the description of the action, such as the action type and the object which is associated with the current cursor position, by means of GetAction command. After the execution of GetAction command, evaluation of the switch statement begins.

Let us assume that a user's action (for example, press of a right button of the mouse) which is associated with "POPUPMENU" is activated. In response to this action, a pop-up menu appears on the display (DisplayMenu command). A menu item selected by the user is then activated.

On the other hand, in the case that the action (for example, a drag operation) specified by the user corresponds to "MOVE", the MoveObject command is first executed. Next IP asks VPSM
main()
{
/* set up of initial display */

while(1)
{
    SN = GetCurrentSceneNumber();
    action = GetAction(SN);
    switch(action.type)
    {
        case POPUPMENU:
            items = create_items(action.object);
            select_item = DisplayMenu(SN, title, items);
            processing(select_item);
            break;
        case MOVE:
            if (movable(action.object))
            {
                shift = action.end_position -
                        action.start_position;
                MoveObject(SN, action.object, shift);
                objects =
                    GetObjects(SN, action.end_position);
                items = InterpretOverlap(objects);
                select_item =
                        DisplayMenu(SN, title, items);
                Execute(SN, action.id, select_item);
            }
            break;
    }
}
}

Figure 7  Description for construction of HI-VISUAL'89

about icon(s) existing at the current cursor position by using the
GetObjects command. The function InterpretOverlap is then
activated to interpret the meaning associated with the
combination of icons. The possible functions the combination
of icons can take are displayed in a menu by using the
DisplayMenu command. One of these functions is selected by
the user and the function module associated with it is then
executed (Execute command).

5. Conclusions

In this paper, we proposed a generic model for
constructing visual programming systems called the Visual
Programming Space (VPS), and a software system to manage
programming facilities of VPS called the Visual Programming
Space Manager (VPSM).

VPS is organized by a sequence of scene frames. A scene
frame manages the information organizing a scene to be
displayed on the screen and actions/operations assigned to the
scene frame.

VPSM provides basic commands for the construction of
visual programming systems. By using these commands, the
system designer can easily develop any specific visual
programming system he/she wishes. We also have presented
HI-VISUAL'89 designed for the programming of office
procedures as an example for explaining how VPSM works in
developing specific programming systems.

The system presented in this paper has been totally
implemented on a workstation Sun-3, under the X-window
system and the UNIX operating system. VPSM itself has been
implemented with C language.

In conclusion, the system we have developed will
certainly provide a friendly programming environment even for
non-programmers. The term 'non-programmers,' however,
does not mean 'beginners.' What is really important here is to
provide a higher degree of friendliness for non-programmers in
an actual programming environment. Professional work will
then be programmed by non-programmers resulting in an as-yet
unpredictable improvement of software quality.

Acknowledgements

The authors wish to thank Y. Tahara, a former graduate
student of Hiroshima University, for his cooperation in the
initial stage of investigating the VPS/VPSM development.
Thanks is also due to the current graduate students, M. Kado
and Y. Nishimura for their contributions in the implementation
of the system.

References

    Programming Environment," IEEE Computer, 17, 11, pp.7-25,
    Nov. 1984.
    Set," Proc., IEEE Workshop on Visual Languages, pp.109-120,
    System for Programming by Demonstration," IEEE Software, 2, 2,
[8] M. Edel, "The Tinkertoy Graphical Programming Environment,
    Proc., IEEE Computer Software and Applications Conf., pp.466-
    Programming System for Children," in Visual Languages, Ed. by
    S. K. Chang, T. Ichikawa, and P. A. Ligomenides, Plenum Press,
    Dynamic Icons," Proc., IEEE Workshop on Visual Languages,