# A Visual Medium for Programmatic Control of Interactive Applications

**Luke S. Zettlemoyer and Robert St. Amant**
Department of Computer Science
North Carolina State University
EGRC-CSC Box 7534
Raleigh, NC 27695-7534
{lszettle | stamant}@eos.ncsu.edu

## ABSTRACT

The VisMap system provides for “visual manipulation” of arbitrary off-the-shelf applications, through an application’s graphical user interface. VisMap’s API-independent control has advantages for tasks that can benefit from direct access to the functions of the user interface. We describe the design goals and architecture of the system, and we discuss two applications, a user-controlled visual scripting program and an autonomous solitaire-playing program, which together demonstrate some of the capabilities and limitations of the approach.

### Keywords

Interaction techniques, agents, demonstrational interfaces, development tools

## INTRODUCTION

In modern software environments, interactive applications often control one another in an arrangement that can lead to increased modularity, improved software reuse, and more coherence in the user interface, among other benefits. Rather than building special-purpose, standalone utilities, a developer can extend an application’s functionality by way of its application programming interface, or API. This approach is followed by many commercial applications such as Netscape Navigator.

Unfortunately, current techniques for the programmatic control of interactive applications have subtle shortcomings. Suppose that I have devised a set of application-independent methods for computer-assisted tutoring for word processing, methods that depend on close interaction (perhaps at the mouse gesture level) with the user. As a developer, I face a number of obstacles. I must either limit my development to a single application or be forced to develop several versions of my software, one for the API of each different application—sometimes for each different version of a single application. I must hope that the developers of each API have had the foresight to support the types of interaction I require. I must hope that the functions in each API are appropriate for the abstractions that will appear in my extensions to the user interface of the application. Even if my project appears trivial, from a user interface design viewpoint, a variety of such technical issues may bring it to a halt.

The problem lies with the mismatch between the functionality of an application as seen through its user interface and as seen through its API. The functionality of an interactive application is most naturally defined by its user interface: its capabilities have been carefully developed to offer specific coverage of tasks, to act at an appropriate level of abstraction, to accommodate the cognitive, perceptual, and physical abilities of the user. The API, on the other hand, is much more closely tied to the software architecture, with only an indirect relationship to the user interface. For some tasks, this indirection rules out the most appropriate means of managing interaction with the user. In some situations, we want to be able to control an interactive application directly, through the same medium that users rely on—its user interface.

We have developed a system, called VisMap (for "visual manipulation"), that supports the control of an application through its graphical user interface, bypassing its API. VisMap takes its input from the screen display, runs image processing algorithms over its contents to build a structured representation of interface objects, and passes this representation to a controller program. Responses from the controller are output by VisMap as mouse and keyboard gestures that control the application. VisMap allows a broad range of interaction with an application in this way, through the same medium as the user.

At first glance this approach may seem a profligate waste of processing power. Consider, however, that much of the time the processor would otherwise sit idle; much of the additional processing cost is hidden. Visual manipulation has potential advantages as well. For various reasons some applications lack an API; others allow only limited control through their API. In some cases (e.g. online tools for layout and task analysis) an API cannot substitute for direct access to the events and appearance of the interface. In general we hope that VisMap will contribute to the development of systems for programming by demonstration, improved macro recorders, wizards, help systems, tutorials, advisory systems, visual scripting systems, and agent-based systems—opportunities to extend off-the-shelf interactive systems that one cannot modify directly as a developer.

VisMap is a relatively new system, and thus we have not yet built applications in all these areas. Instead, we describe two applications that give the flavor of the approach, demonstrating its feasibility and some of its generality. The first, VisScript, gives users a simple facility for running visual scripts. Though as yet a simple prototype, VisScript promises greater flexibility and coverage than existing macro definition utilities. The second application, VisSolitaire, allows an artificial intelligence planning system to play solitaire. As a problem domain, solitaire poses few conceptual difficulties; rather, the task highlights VisMap's ability to control an application that has non-standard interface controls and no API. In both applications, VisMap is responsible for low-level interaction with the application, while the relevant domain knowledge is provided by easily interchanged controller modules.

Our work benefits the CHI community in two ways. From a developer's perspective, the direct benefit is a flexible complement to API-based control of interactive applications. Imagine for example building a tutorial or walkthrough for an arbitrary suite of applications, including tasks in the operating system, and being able to work at a consistent level of abstraction and with the same vocabulary across all the diverse components of the system. A designer need not be constrained by software architecture limitations when tasks can be accomplished through the user interface. The potential benefit for users is equally great, if less direct. The potential applications for VisMap, as described above, can extend the functionality and coherence of direct manipulation interfaces. Our early experience with the system and its applications has shown the approach to have considerable promise.

## DESIGN GOALS

VisMap acts as an intermediary between a controlling application and an application to be controlled, which we will call the "controller" and the "application" respectively. Informally, VisMap provides the controller with the eyes and hands (the sensors and effectors) necessary to manage the application. The user may even act in the role of a controller, when appropriate. An earlier version of VisMap [13] has given us a good deal of insight into the design goals for this kind of system. Three sets of goals arise from the need to interact with applications, controllers, and users.

An enormous effort goes into the development of application user interfaces [6], toward the implicit goal of matching the abilities and limitations of human users. In interacting with applications through such interfaces, the ideal system accommodates and exploits this bias toward human-like perception, action, and cognition wherever possible.

1. Sensors: At the "physical" level, the system must process input from the joint human-computer system. This includes monitoring the mouse and keyboard as well as distinguishing visual and temporal patterns in the contents of the screen.
2. Effectors: At the same level, the system must be able to control an application through its user interface, via mouse and keyboard gestures.
3. Information processing: The system must be able to recognize the patterns in its input stream as constituting specific types of information, and to combine these patterns into known structures and relationships.

A second set of goals arises from the need to support controller programs. From a controller's perspective, the ideal system has these properties:

4. Coverage: It must provide the functions necessary to control a variety of applications, but in an application-independent manner.
5. Extensibility: It must support extensions, possibly application-dependent, beyond the basic coverage functions.
6. Representational flexibility: It must support a means of adjusting the amount and level of detail—setting the appropriate level of abstraction—in the information exchanged.

Finally, the ideal system cannot neglect the user, who is interacting with an otherwise direct manipulation environment. The system adds an element of autonomy to the environment: it may in some cases take actions not explicitly specified by the user. While this can be managed without subverting the benefits of direct manipulation [11], the ideal system must at a minimum address these issues:

7. User control: The system must respond continuously to user control, when it is available.
8. User awareness: It must be clear at all times whether the system is taking autonomous action in the interface.

The system we present in the next section does not meet all of these goals; it does however approach our ideals in its design. Even an ideal system, however, will encounter several limitations. First, the benefit of API-independence is offset by dependence on an application's user interface. If an interface supports extreme variations in look and feel layered over the same functionality, this can result in less generality for a visual system rather than more. Second, a purely visual system will have no access to the internal data

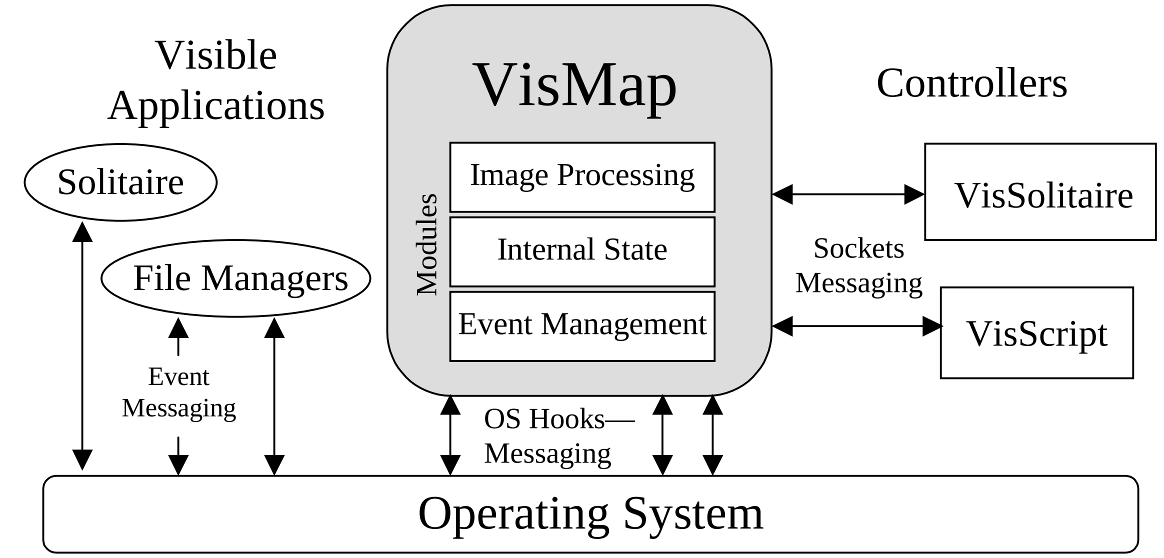


**Figure 1. VisMap General Architecture and Communication**

structures of an application. Others have demonstrated [9] that a good deal of information can be derived directly from the screen, information that would be difficult to gain otherwise, but not all useful information appears on the screen. Third, a visual system has no choice but to work through the interface. Even if a task might be better carried out behind the scenes, without the user's knowledge, this option is not available. These implications are unavoidable for a visual system and limit the range of its application.

## SYSTEM DESIGN

We have described the general architecture of VisMap elsewhere [14]. Here we give a brief recap and discuss how it meets the design goals identified in the last section. VisMap is divided into three separate modules: the event management module, or EMM, the image processing module, or IPM, and the internal state representation module, or ISRM. These three modules provide all of VisMap's processing. VisMap's architecture and methods of communication can be seen in Figure 1.

The EMM handles events as both a sensor and an effector. It manipulates the operating system's event queue, able both to observe user-initiated events as they pass through the queue and to insert its own events into the queue. In its sensor role the EMM supports the first design goal by maintaining an internal variable-length queue of user-generated events for processing in later stages. In its effector role the EMM meets the second design goal: its event insertions are indistinguishable from user-generated events. The EMM can be used to select icons, click buttons, pull down menus, turn on radio buttons, and carry out all other standard operations we have experimented with. These are implemented as sequences of low-level events: *move-mouse, mouse-down, mouse-up, key-down,* and *key-up*. Some useful higher-level abstractions, such as *click-button*, which requires a sequence of these more primitive events, have been implemented, but the issue of deciding on an appropriate level of abstraction currently remains open.

The IPM rounds out support for the first design goal and partially meets the third, in a conventional sequence of image processing stages [1]. The IPM begins with a two-dimensional image of the screen. In the *segmentation* stage, the module breaks the image into pixel groups by color. The white background of a list box, for example, would end up in a single group. In the *feature computation* stage the module attaches features to each group that describe its internal structure and its relationship with other groups. Figure  gives an example that shows how the "area" feature of an pixel group would be computed. Note that these computations are data-driven, bottom-up—there is no guarantee that a feature will be useful for the interpretation of a given group. In the *interpretation* stage, features are iteratively combined via rules to build structures that correspond to "meaningful" objects. In contrast to the second stage, interpretation is top-down. Rules are hypotheses that must be verified in their identification of objects in the interface. Figure  shows an interpretation rule for identifying a list box.

The ISRM is responsible for integrating the information provided by the IPM and the EMM over time. It maintains a representation of the temporal and spatial changes observable through the screen buffer. This information is then available to controllers so they can observe changes in their applications. The ISRM completes our coverage of the sensor/information processing design goals.

*Operation GetArea()*
*MaxPossibleNumPixels = GetWidth() * GetHeight()*
*Area = ActualNumPixels() /MaxPossibleNumPixels*
*Return Area*

**Figure 2. A feature computation of area**

*If there exists a downArrow()*
*That is containedIn() a raisedButton()*
*That is toTheRightOf() a rectangularTextArea()*
*Which is recessed() and has a width()*
*greater than its height()*
*Then we have found a list box*

**Figure 3. An interpretation rule to identify a list box**

In pursuing the first set of design goals we have in effect defined a simple artificial user, a kind of programmable user model. A system limited to our description so far, however, is incomplete: it is entirely independent of an operating environment. This issue is addressed by the second set of design goals, which require that we flesh out the feature computation and interpretation rule libraries of the IPM until they have sufficient coverage of functionality in a real user interface (Microsoft Windows in our current implementation.)

The IPM contains in total 29 feature computation functions and 80 interpretation rules of the types shown in Figure 2 and Figure 3. A sample of the IPM's processing is shown in Figure 4. The top picture shows the original interface, the bottom picture all of the widgets that the IPM has identified. Given the performance of its libraries across a variety of applications, VisMap can claim good coverage (the fourth design goal) in interacting with the user interface.

The fifth design goal requires that a controller be able to extend VisMap's capabilities to handle special-purpose processing. For example, an application may include specialized controls that are not commonly found in other domains and are not be available thorough any APIs. Server-side image maps displayed in web browsers are a common example. Visual representations of interactive widgets are not accessible to the browser or the local system; processing is handled remotely by the server. To a VisMap controller, however, a button graphic with the appropriate appearance, however generated, is treated no differently than an actual widget in a local application.

The sixth design goal entails giving a controller the ability to tailor its interaction with VisMap to an appropriate level of abstraction. For example, should every mouse movement event, every mouse up and mouse down, be passed to the controller? Perhaps common abstractions, such as selection? The current implementation of VisMap is relatively inflexible in this regard. The level of representation is programmable, but cannot be varied at run time. Controllers connect to VisMap through standard TCP sockets to communicate with a fixed set of commands and responses. The interaction, though limited, supports the necessary range of communication for our prototype controllers.

To summarize VisMap's coverage of the design goals up to this point, the sensor/effector design goals are met. VisMap can reliably recognize all the user interface controls we have worked with: buttons, scroll bars (including the scroll box, scroll arrows, and background regions), list boxes, menu items, check boxes, radio buttons and application windows. VisMap also meets the fourth and fifth design goals by providing a basic set of functions, which can be extended at the cost of a nontrivial programming effort. The sixth design goal of variable abstraction is not met.

User interaction issues, touched on in the final two design goals, raise a number of unsolved problems. VisMap essentially adds another player to the user interface environment. Depending on the controller, a VisMap-based system may exhibit a high degree of autonomy or none at all. (Examples of these two extremes are described in the sections below.) Mixed-initiative interaction with an automated system raises a number of elementary HCI questions: Will users know where they are in the interaction

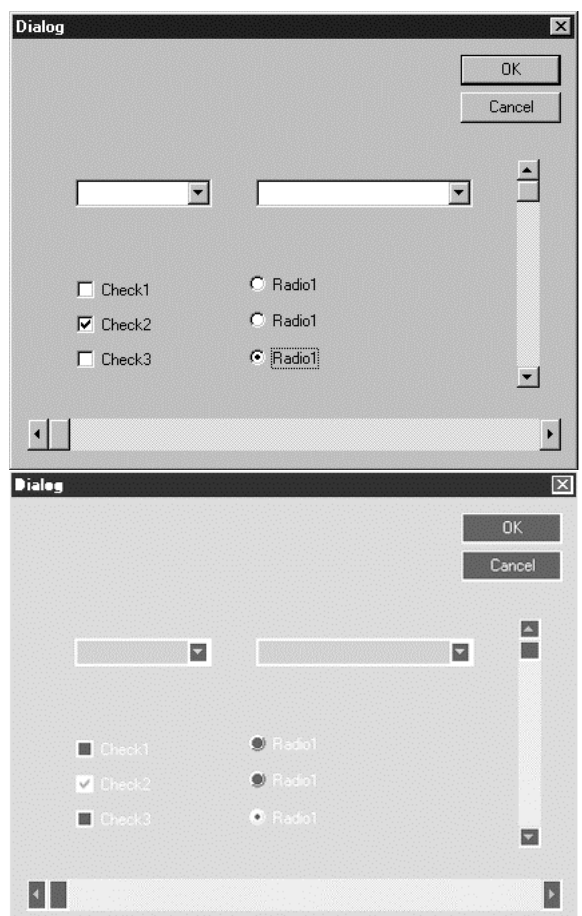


**Figure 4. IPM results**

| Screen Resolution | Sample Execution Time |
|---|---|
| 640x480 | 2.32 seconds |
| 800x600 | 3.72 seconds |
| 1024x768 | 5.85 seconds |
| 1152x864 | 7.81 seconds |

**Figure 5. Screen Processing Execution Times**

process? Will they know how they arrived there, what they can do, where they can go from there? These questions cannot be answered for VisMap in isolation, but must be considered in the context of its integration with a specific controller and application.

In VisMap's current state of development, we must sidestep these and related questions until we have gained more experience with its use. VisMap does nevertheless ensure that its autonomous actions are accompanied by strong visual cues that control has temporarily shifted away from the user, and that its activity can easily be interrupted and turned off.

VisMap comprises 2,800 lines of C++ code. Runtime efficiency has been considered for different screen resolutions and effort is being put into increasing the speed of execution. Current sample running time for various screen resolutions on a 300 megahertz Pentium II processor with 128 megabytes of RAM are given in Figure 5.

Notably, only about 3% of the system is specific to the Windows operating system. This lends support to our claim that VisMap is largely platform-independent and operating system-independent (e.g., VisMap should easily port to a Windows emulator running on a Macintosh or a Unix machine.) Another 40% is specific to Windows interface conventions for visual display: the appearance of buttons, list boxes, and other controls. This latter point may seem to contradict our earlier claim; however, one of the strengths of VisMap is that it separates operating system issues from user interface issues. A port to the Macintosh and its native environment, to test the degree of dependence on the user interface, is in planning.

### VisSolitaire

Building a system that allows a computer to play solitaire without the intervention of a human user—in other words, to waste time all by itself—has more serious underpinnings than it might initially appear. A great deal of work in agent-based systems, demonstrational interfaces, and other AI-driven approaches to improving user interaction could benefit from a stronger connection to commercial applications, to gain leverage from market pressures that constantly increase the power and flexibility of interactive software. Many (even most) interactive AI systems, however, never leave the research laboratory.

We believe that a contributing factor is the difficulty in developing a tight integration with existing applications at the user interface. Solitaire represents applications that pose obstacles to such an integration:

- The application uses non-standard icons in its interface, which means that a controller cannot simply ask for, say, the positions of the windows or buttons in the interface.
- The application has no API, which means that conventional programmatic control is not possible in any case.
- The internals of the application are not available to us as developers; we cannot simply rewrite it to accommodate external control.
- Assistance in the application can reasonably take the form of direct action, rather than advice to the user (e.g., "In order to accomplish your task, follow steps X, Y, and Z.")

VisSolitaire, an exemplar of a VisMap-based system, has three components. The first component is the application, an unmodified version of Microsoft Solitaire. VisMap is the second component, responsible for the visual and physical aspects of the game, such as interpreting layout and screen icons, and moving the mouse. The third component is an AI planning system (UCPOP [8]) which handles the strategy of solitaire through an abstract game-playing representation.

The integration of these components is straightforward. The application generates an initial game state, displayed as card images the screen. For the initial move and each thereafter, VisMap identifies the cards and groups them in their layout, the stock, waste, tableau, and foundation piles. This process occurs through the segmentation, feature computation, and interpretation stages described above; it leads to a screen-coordinate representation of all cards in play. From the Cartesian representation VisMap constructs a symbolic abstraction and passes it to the planner. The planner processes the game state, selects a move, and passes it back to VisMap to be executed.

The planner maintains most of the relevant knowledge about the problem, represented in a set of plan operators, or a domain. The planner analyzes the state representation supplied by VisMap and constructs a plan to satisfy the top-

```
(:operator tableau-to-foundation
  :parameters (?tn ?tr ?s ?fn ?fr)
  :precondition  (and (tableau-last ?tn ?tr ?s)
                      (foundation-last ?fn ?fr ?s)
                      (previous-rank ?tr ?fr))
  :effect  (tableau-to-foundation ?tn ?tr ?s ?fn ?fr))
```

**Figure 6. Solitaire operator**

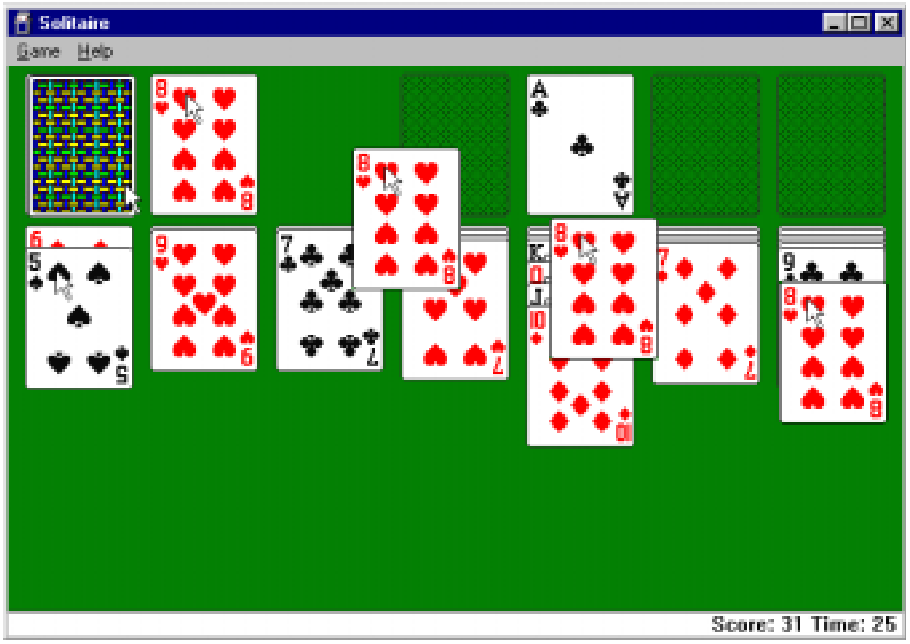


**Figure 7. VisSolitaire playing solitaire**

level goal of making a move. In actuality, because we are not especially concerned with solitaire-playing strategies, there is very little planning involved. A sample operator, of ten in total, is shown in Figure 6. Parameters in the operator shown contain numerical identifiers for the tableau and foundation piles and the suits and ranks of the cards. If the precondition of this operator holds in the current environment, the effect specifies that the card of suit *s* and rank *tr* should be moved from tableau pile *tn* to foundation pile *fn*, to end up on the card with suit *s* and rank *fr*. The top-level goal for the planner, in all initial states, is simply the disjunction of the effects of all its operators.

In this implementation, the planner returns operators that are specified down to the level of commands to press and release the mouse button and to move the mouse from one location to another (though these locations are in an abstract representation independent of screen coordinates.) We could easily have arranged for interaction to occur at a higher level of abstraction: "Drag 4S to 5H," for example, ignoring the lowest level of mouse event processing, or even "Move 4S to 5H," abstracting away the relationship between mouse gestures and card movement altogether. Our decision was to retain a high degree of detail at the planner level, rather than adding what could be considered domain knowledge to VisMap. A sample interaction sequence between VisMap and VisSolitaire is shown in Figure 8. An important issue remains open: how the level of abstraction of the interaction can be modified, ideally on the fly, for conceptual clarity and efficiency.

VisSolitaire plays a reasonable game of solitaire, from the starting deal to a win or loss. The planner maintains a minimal amount of state information between moves, including a record of the sequence of its moves. On encountering the same cards after working through the stock, with no intervening moves that have changed the tableau or foundation, the system stops with the loss.

VisSolitaire is implemented in Harlequin Lispworks and communicates with VisMap via sockets. The VisMap feature recognition rules required some time and effort to build, enough to motivate future work on support tools for their development.

### VisScript

Researchers on both sides of the direct manipulation/ autonomous agents debate recognize the importance of visual scripting to the future of direct manipulation interfaces. Shneiderman calls graphical macro tools his favorite project to advance general computing [12]. Myers describes a wide range of benefits to incorporating scripting into the interface [6]: the automation of repetitive tasks, the addition of useful levels of abstraction, the delegation of

VisSolitaire COMMAND:

(GET-CARDS-LAYOUT)

VisMap RESPONSE:

(((8 :HEARTS) 2 1)

((:ACE :CLUBS) 5 1)

((6 :DIAMONDS) 1 2)

((5 :SPADES) 1 2)

…

((9 :CLUBS) 7 2))

**Figure 8. Sample interaction sequence**

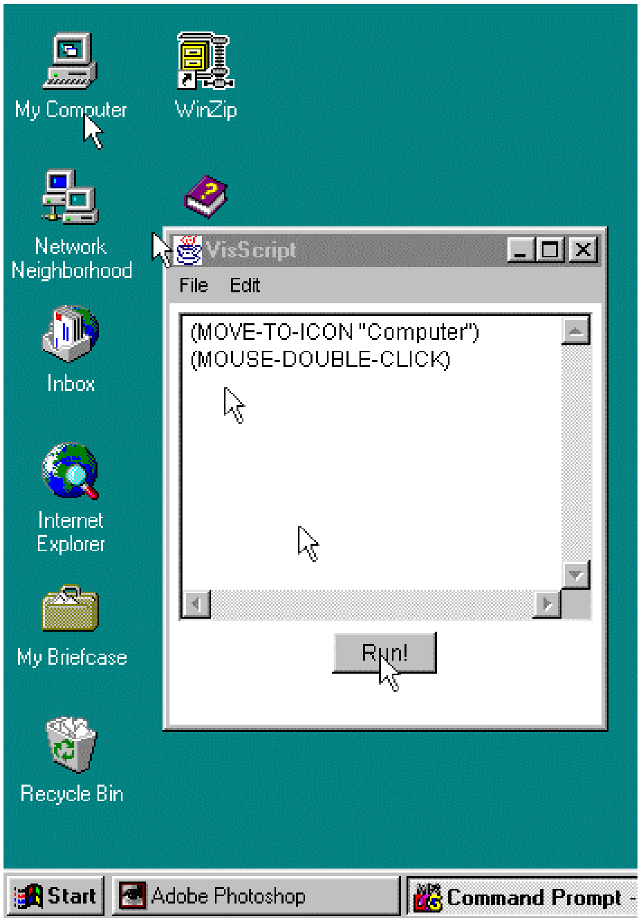


**Figure 9. VisScript executing a simple script**

responsibility for low-level activities. Unfortunately, a visual scripting tool based on current technology, even if it were able to provide all these benefits, would still suffer a number of drawbacks:

- Application-dependence: Some scripting tools can only be used within a single application (e.g. macro definition in Microsoft Word.)
- System-dependence: Some tools can exist only within a specialized interface framework (such as Garnet or Amulet [7].)
- Interoperability: Existing scripting tools that can move between the interfaces of different applications, as well as the operating system, can access only a limited subset of the available functionality.

Another necessary element of a scripting tool is user control—users should be able to write scripts and execute them on demand. We have designed these considerations into VisScript, an early prototype for executing scripts through the user interface. VisScript is an attempt to provide a tool with which users can simplify their interactions with standard pre-existing user interfaces. While VisScript is not a general purpose visual macro system, we have taken important first steps towards reaching the above goals.

VisScript allows users to enter commands and add them to a script to be executed. The current list of commands includes *move-mouse*, *singe-click*, *double-click*, and *move-mouse-to-text*. These commands are combined in Figure 9, which shows a script that allows the system to open a file manager for the top level directory. During execution, the progression through the script is entirely linear; VisScript does not incorporate programming constructs to control its flow. The commands are sent to VisMap to be executed one at a time and the user can watch as the are performed.

VisScript is implemented in Java and communicates with VisMap through TCP sockets. VisScript can run on the same machine as VisMap or remotely. Working with VisMap as a foundation, we were able to develop VisScript in less than two days of programming effort. We consider this evidence of the generality of VisMap and its potential for building other useful tools.

## RELATED WORK

A recent paper describes an earlier prototype of the VisMap system, along with an application in usability testing [14]. The system presented at that time had a number of limitations that are addressed in the current version. The most significant step forward is conceptual: the earlier system presented evidence that a visual manipulation system *could* be built; in this paper we have presented our perspective on how a visual manipulation system *should* be built. More concretely, unlike the earlier system, the current system can run fast enough to handle interaction with users, although not at high rates of speed. Its interpretation rules encompass a broad range of patterns that appear in the user interface, not simply limited to standard controls. It supports multiple simultaneous controllers, for a planned application in cooperative computing environments.

The VisMap effort draws on three main areas of research: user interface agents, programming by demonstration, and programmable user models.

Lieberman outlines a number of areas relevant to the VisMap approach [4]. His discussion emphasizes the importance of granularity of event protocols, styles of interaction with the user, and parallelism considerations. Event granularity determines the level of abstraction at which an agent interacts with an interface. For example, should mouse movements be included in the information exchanged? If not all mouse movements (possibly a very large number, depending on the sampling rate), then which ones are important? An interaction style describes the way in which an agent interacts with the user. That is, it may not always be sufficient for an agent to execute commands in an interface; it may be necessary to communicate directly with the user. This can force a different interaction style, for example, on an agent designed mainly for direct manipulation interactions. Issues of parallelism can enter the picture when the agent and the user both try to manipulate the same interface object. System performance can also be affected by the activities of an agent. As discussed earlier, VisMap does not address these issues in detail. For its current applications, it works at a system event granularity, though its controllers operate at a higher level of abstraction. As yet it has no mechanisms for communicating directly with the user or managing parallel activities.

Potter's TRIGGERS system [9] is an early example of an approach similar to ours. TRIGGERS is an example of a system for programming by demonstration, one of only a few examples that work with off-the-shelf software. TRIGGERS performs pattern matching on pixels on the computer screen in order to infer information that is otherwise unavailable to an external agent. A "trigger" is a condition/action pair. Triggers are defined for such tasks as surrounding a text field with a rounded rectangle in a drawing program, shortening lines so that they intersect an arbitrary shape, and converting text to a bold typeface. The user defines a trigger by stepping through a sequence of actions in an application, adding annotations for the TRIGGERS system when appropriate. Once a set of triggers have been defined, the user can activate them, iteratively and exhaustively, to carry out their actions. From TRIGGERS VisMap adopts the notion that the screen itself is a powerful source of information for an agent, if it can be properly interpreted.

The third area, programmable user models, has contributed only indirectly to VisMap's development. In Young's original description [13], PUMs were engineering models, not to be executed directly. The intention was to provide designers with an engineering model that could give predictions at an early stage in user interface development. This approach has shown significant promise, especially in the recent work of Kieras and Meyer [3]. A natural extension, which VisMap pursues, is the construction of executable PUMs that can be applied directly to implemented systems as well as those in the design stage. The architecture of VisMap has no strong foundation in cognitive theory, but could accommodate such a foundation in an appropriate controller.

## CONCLUSION

We view our work as facilitating technology. Many of the most interesting extensions of graphical user interfaces have been demonstrated in isolated research systems, and have failed to make the transition to commercially available software. We believe that the general layer VisMap provides will allow such work (e.g. in visual scripting [7], demonstrational interfaces [6], mixed-initiative interfaces [10], and agents that interact directly with users [5]) to reach the mainstream.

## ACKNOWLEDGMENTS

We wish to thank Derrick Foley, who contributed significantly to the development of character recognition rules in VisSolitaire. Support for this work was provided by North Carolina State University and the William R. Kenan Institute for Engineering, Technology, and Science.

## REFERENCES

1. Gentner, D., and Nielsen, J. The Anti-Mac Interface, *Communications of the ACM*, 39:8 (August, 1996), 70-82.
2. Gonzales, R.C. and Woods, R.W. *Digital Image Processing*. Addison-Wesley, Reading, MA. 1992.
3. Kieras, D. and Meyer, D. E. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*.
4. Lieberman, H. Integrating User Interface Agents with Conventional Applications. *Proceedings of Intelligent User Interfaces '98.* (San Francisco, CA, January, 1998.) ACM Press, 39-46.
5. Maes, P. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37:7, (July 1994), 31-40.
6. Myers, B. Demonstrational Interfaces: A Step Beyond Direct Manipulation, *Watch What I Do: Programming by Demonstration*, Allen Cypher, et. al., eds. MIT Press Cambridge, MA. 1993. pp. 485-512.
7. Myers, B. Scripting Graphical Applications by Demonstration. Proceedings of CHI '98. (Los Angeles, CA, April, 1998.) 534-541.
8. Penberthy, J. and Weld, D. UCPOP: A sound, complete, partial-order planner for ADL. *Proceedings of the Third International Conference on Knowledge Representation and Reasoning.* 1992. Morgan Kaufmann, 103-114.
9. Potter, R. Triggers: Guiding Automation with Pixels to Achieve Data Access. In *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA. 1993.
10. Rich, C. and Sidner C. L. Adding a Collaborative Agent to Graphical User Interfaces, *Proceedings of UIST'96*, (1996), 21-30.
11. Shneiderman, B. Direct Manipulation for comprehensible, predictable, and controllable user interfaces. *Proceedings of Intelligent User Interface'97.* (Orlando, FL, January, 1997.) ACM Press, 33-39.
12. Shneiderman, B., and Maes, P. Debate: Direct Manipulation vs. Interface Agents. *Interactions*, 4:6 (November and December, 1997), 42-61.
13. Young, R. M., Green, T. R. G., and Simon, T. Programmable User Models for Predictive Evaluation of Interface Designs. *Proceedings of CHI '89*. 15-19.
14. Zettlemoyer, L. S., St. Amant, R., and Dulberg, M. S. Application control through the user interface. *Proceedings of Intelligent User Interfaces '99.* (Redondo Beach, Los Angeles, CA, January, 1999.) To appear