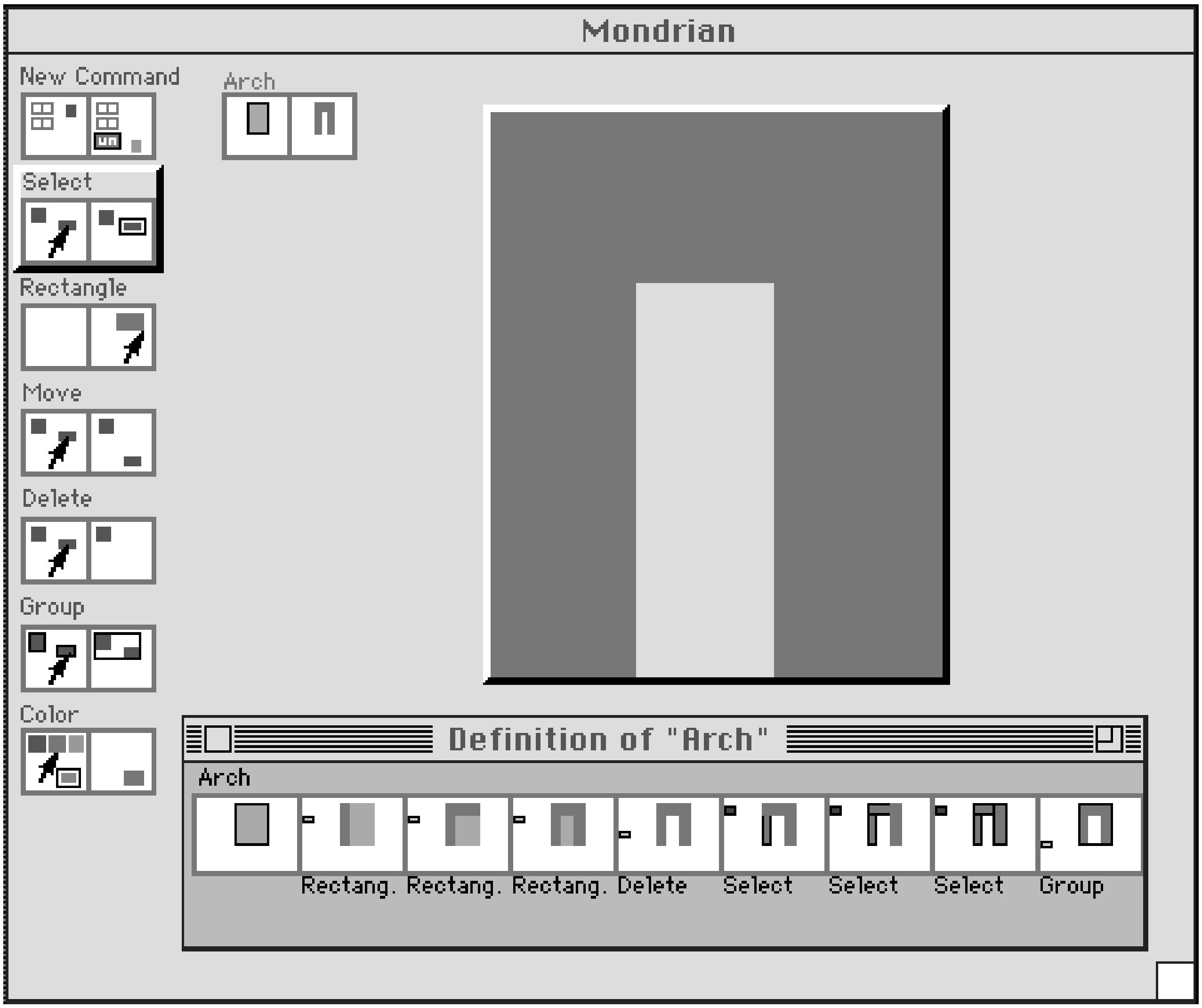

Mondrian
New Command
Select
Rectangle
Move
Delete
Group
Color
Arch
Definition of "Arch"
Arch
Rectang.
Rectang.
Rectang.
Delete
Select
Select
Select
Group

Mondrian: A Teachable Graphical Editor

Chapter **88**

by
Henry Lieberman
Media Laboratory
Massachusetts Institute of Technology

Introduction

Mondrian is an object-oriented graphical editor that can learn new graphical procedures through programming by demonstration. A user can demonstrate a sequence of graphical editing commands on a concrete example to illustrate how the new procedure should work. An interface agent records the steps of the procedure in a symbolic form. Machine learning techniques, track relationships between graphical objects and dependencies between the interface operations. The agent generalizes a program that can then be used on "analogous" examples. The generalization heuristics set it apart from conventional "macros" that can only repeat an exact sequence of steps. The system represents all operations using "storyboards" which depict examples. By bringing the power of procedural programming to easy-to-use graphical interfaces, we hope to break down the "Berlin Wall" that currently exists between computer users and computer programmers.

Conventional Programming Is A “Berlin Wall” Between Users And Applications

Microcomputers such as the Macintosh and IBM PC enable first-time computer users to effectively use such applications as desktop publishing and spreadsheets. But the activity of writing computer programs still remains inaccessible to the vast majority of users.

While modern user interfaces represent objects of interest in the application domain as directly manipulable graphical objects, extending the interface by adding new commands currently requires mastering a programming language such as C or Lisp. To write a program, the programmer must work in a text-based environment completely divorced from the graphical interface. The barrier between the graphical interface and programming environment is a kind of "Berlin Wall" that prevents users from getting full control over their applications.

An alternative to programming in a conventional textual language is to design an interface that can be extended with new operations directly through interaction with the interface itself. The interface can incorporate a learning capability that can record user interactions, using them as the basis for defining new operations. The user *teaches* or *instructs* the interface rather than “programs” it.

Bringing Programming To "Visual Thinkers"

Our intended target users are "visual thinkers": graphic designers, interface designers and multimedia designers who are currently disenfranchised by text-based programming environments. First, this community will provide a good test of accessibility, since people with visual design backgrounds often have difficulty learning conventional textual programming languages. Though many designers now use graphical editors, as their work becomes increasingly ambitious, these people have a need for complex interactive control in their customized applications, but currently find programming beyond their reach.

Studies of the design process in these fields show that the primary method of conceptualization is the generation and critique of concrete visual examples. [Vertelney, Arent and Lieberman 89] reports a study of one encounter between the visual design and computer science perspectives on the design of a particular interactive interface. As communications media become more interactive and programs deal more and more with graphics and dynamic objects, the programming and visual perspectives will inevitably converge. Thus programming by demonstration should be particularly congenial to the kind of synthesis between programming and visual perspectives that we will need for interactive graphical interfaces in the future.

Mondrian is a simple object-oriented graphical editor, in the style of MacDraw, whose interface can be extended with new graphical primitives and procedures by demonstrating sequences of actions on a concrete example. An interface agent records the steps, and generalizes a program that can be used on "analogous" examples in the future. The interface agent also provides feedback to the user about what has been learned.

Programming By Demonstration = Macros + Interface Editors + Generalization

One way to understand the idea of programming by demonstration is to see that it integrates three kinds of existing applications, each of which is useful but has some serious shortcoming in its present form.

“Macro recorders” such as Macromaker, Tempo II, HP New Wave, and those found in applications such as Excel have an interface mode that records user actions such as coordinates of mouse selections and typing. These can be played back at some later time to repeat the sequence of operations. The original actions serve as an example, which can be repeated on different data.

But these kinds of macros are brittle. They are usually limited to exact repetition of the sequence of operations on which they are defined. They are very sensitive to irrelevant details of the interface environment, such as position of icons and windows, and sometimes even timing. Some authors [such as Kurlander in this volume] allow the term “macro” to refer to more general recorded programs, but the current commercial “state of the art” seems limited to linear, literal recorded sequences of user actions.

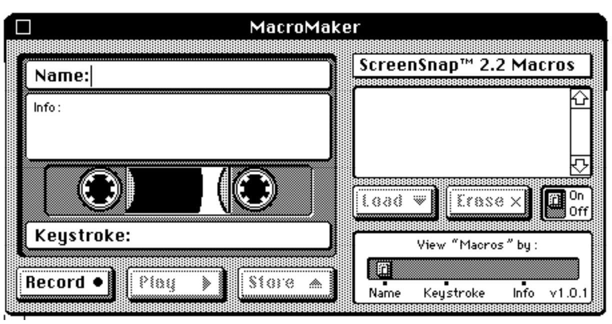


Figure 1: Macromaker

Interface editors such as Hypercard, the NeXT Interface Builder, and Macintosh Lisp's Interface Tools allow graphical editing of the position and size of objects representing interface components such as buttons and text fields. A working interface is constructed by graphically editing examples of the interface's appearance. But such systems are limited to piecing together previously defined behavioral components. They cannot introduce new behavior, except by making connections to code modules programmed in a conventional textual programming language.

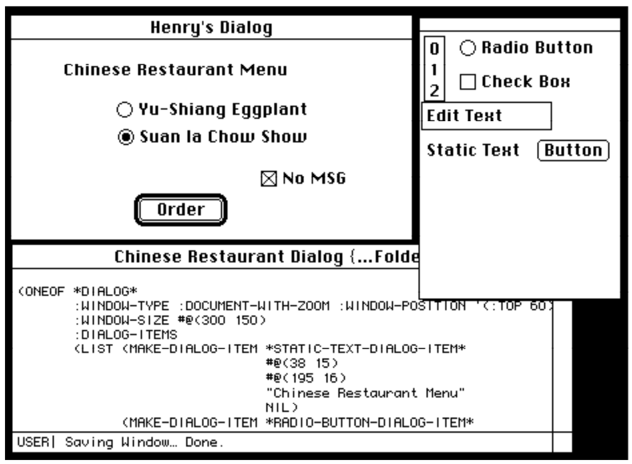


Figure 2: Macintosh Common Lisp Interface Tools

Generalization techniques from artificial intelligence have the capability to infer generalized procedures or descriptions from concrete examples. These include Winston’s arch learning program, xplanation-based generalization and case-based reasoning. However, the interfaces to these programs have all depended solely on typed descriptions of the examples. AI has missed an opportunity to apply machine learning in the graphical interface domain. The machine should learn to construct an interface from watching example interactions with the user, and perhaps from user advice about how the examples should be generalized.

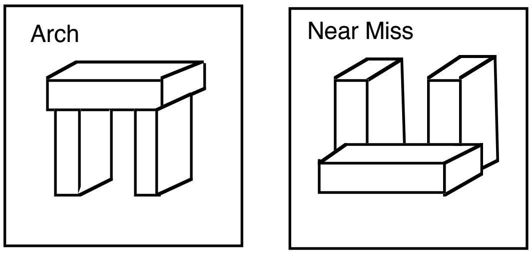


Figure 3: Winston's Arch learning program

Programming by demonstration combines the best of these techniques. User actions are recorded in a symbolic form that does not depend on details such as screen coordinates. Generalization of the program removes the literal playback constraint that macros have. Programming by demonstration records the procedures that drive the interface rather than just edit the properties of interface objects as graphical interface editors do. Programming by demonstration brings to graphical interfaces some of generalization power of AI learning programs.

Programming by demonstration introduces a teaching metaphor into the programming process. The programmer plays the role of a teacher, the computer a (very dumb) student. Good teachers know that the best way to convey an idea to a student is through a set of well-presented examples, and enough advice to enable the student to generalize his or her experience to new examples in the future. Since people teach and learn most effectively via examples, why can't examples serve as a means for teaching machines how to perform procedures?

Dominoes: Example-Oriented Icons

While the interface of Mondrian strongly resembles other object-oriented graphical editors such as MacDraw, an unusual aspect is its icons. Each Mondrian icon is a *domino*, consisting of two linked rectangular parts. The domino is also labeled with the name of the command, although the purpose of the command should be apparent just from the pictorial information alone.

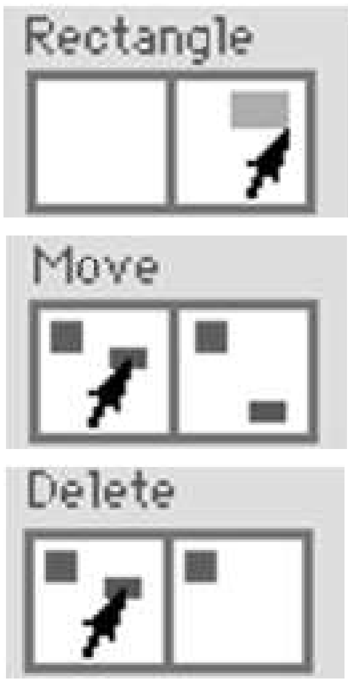


Figure 4: Some of Mondrian's domino icons

The domino is a visual representation of an example of the use of the command. The left and right sides of the icon are reduced-size *before and after* panels, a snapshot of the screen just before the command is executed, and a snapshot of the screen just after the command is executed. Using these before and after pictures to illustrate the built-in commands is a good way of encouraging the user to think about representing operations by their effect on concrete examples.

For example, the for the command that creates a new rectangle consists of a blank screen for the "before" picture, and a screen containing the newly created rectangle as the "after" picture. The icon for the delete command shows one of the visible rectangles selected in the "before" picture, and absent from the "after" picture.

The details of the screen snapshots can subtly indicate program state associated with the command. If the default drawing color is changed, the color of the new rectangle in the "after" part of the domino is changed to match. This provides feedback to the user that rectangles will now be drawn using the new default color.

Sometimes the screen snapshots are abstracted, not exact replicas of a screen state. Some aspects may be omitted, others emphasized, to better communicate the effect of the command. Examples of visual abstraction are enlarging the cursor to emphasize its position, or replacing the details of a bitmap image by its outline to indicate its size and position.

An Example: Learning How To Draw An Arch

I will now present an example of how to extend Mondrian's interface through programming by demonstration. Initially, the only graphical object creation primitive provided in the Mondrian editor creates colored rectangles. We can teach the system how to create a new kind of graphical object built out of sequences of graphical editor commands, providing, of course, that it can be made with the existing graphical vocabulary. The particular example I have chosen here is to teach the system how to draw an *arch*, in honor of Patrick Winston's learning program that could be taught a generalized definition of an arch by presenting examples [albeit non-graphical examples].

The arch primitive will accept as its single argument a rectangle to serve as a *template*, in which the arch will be inscribed. To indicate this, an example of the template is selected, and the New Example icon chosen. Selecting an argument to an operation being defined instructs the system to look for relationships between the argument object and any objects that are created or selected in the course of demonstrating the operation. A command may be given more than one argument, and the order of arguments is considered to be significant.

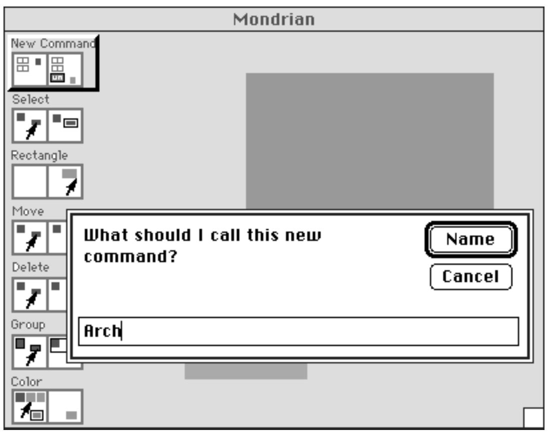


Figure 5: Naming a new command

In the upper left-hand corner is the New Command icon, which initiates the definition of a new command. The before and after pictures of the New Command icon show a new domino icon being added to the set of available operations. Choosing the New Command icon causes a question mark to appear in the after picture. This indicates that the system is in "remember mode", recording the user's actions.

The system asks the user to type a name for the new command. Then it manufactures a new domino icon to represent the new command being defined. This icon has a "before" picture consisting of a tiny copy of the state of the screen at the time the New Example operation is invoked, and a question mark for the "after" picture, since the situation after execution of the command is not yet known.

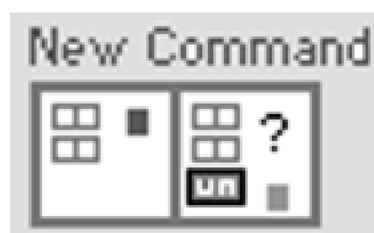
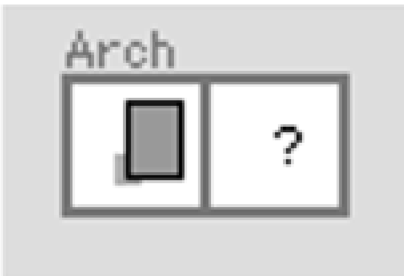


Figure 6: New command icon

Figure 7: Arch icon just after the start of the definition



The "before" picture captures the entire state of the screen, even including objects that were not indicated as input arguments to the command being defined. Though these objects may be irrelevant to the actual working of the operation, including them helps establish context for the example. If the visual complexity of the icon becomes a problem, these extraneous objects could easily be omitted to simplify the picture.

The appearance of the icon for the new operation at the start of the definition of the command is important, because it affords the opportunity to invoke the new operation itself in the middle of its own definition. The call to the new operation is itself recorded as part of its own definition. This will become essential for defining recursive commands, as in the author's Tinker system [Lieberman 84, 87].

Now, we demonstrate to Mondrian how to draw the arch. We draw rectangles for each of the pillars of the arch, and for its horizontal top portion. The pillars and top of the arch are inscribed using the corners of the template rectangle as a guide. We needn't, however, match these points exactly, because Mondrian has a kind of *semantic gravity* that will tolerate small errors in alignment.

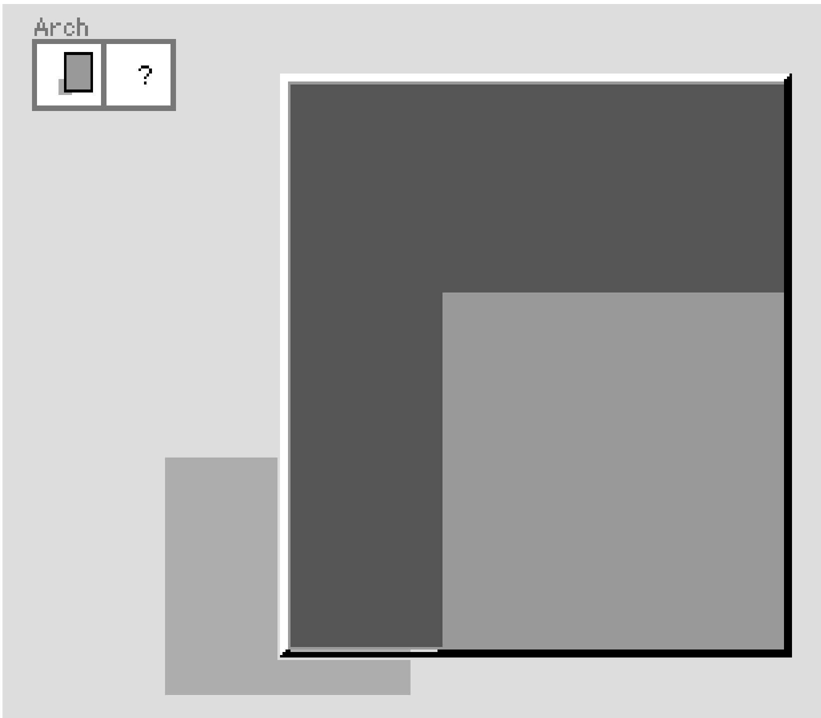
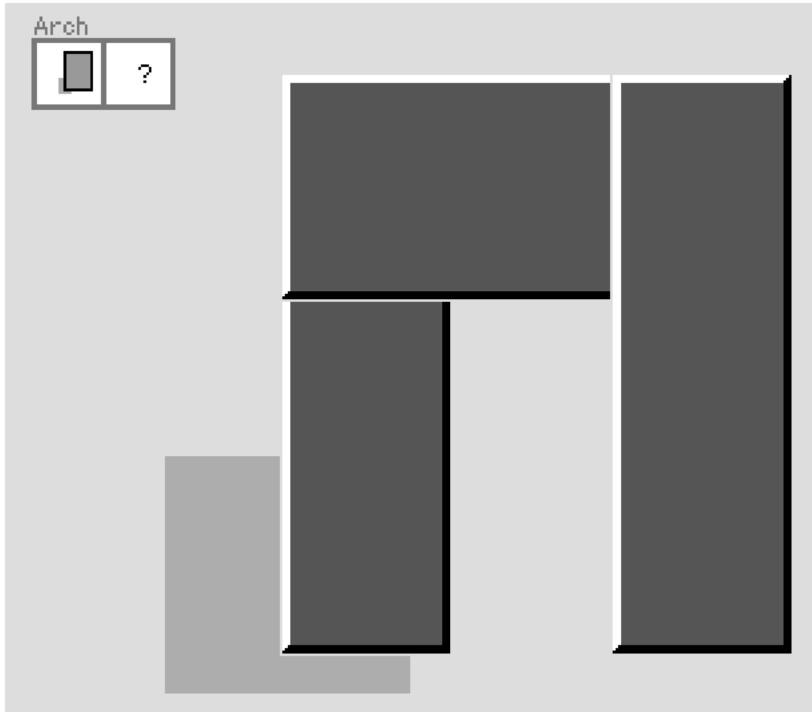


Figure 8: Drawing the Arch

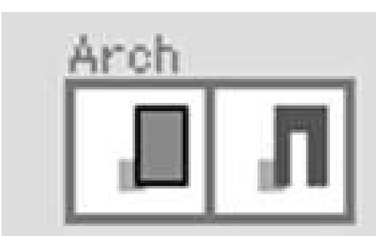
We continue defining the arch. We no longer need the original template rectangle, so we delete it. We now have three separate rectangles that form the arch, but what we really want is a single object. So we do a multiple-select that includes the three rectangles, and then the Group operation, making the three rectangles into a single object.

Figure 9: Selecting the three rectangles that make up the Arch



This concludes the definition of the arch. Clicking on the New Command icon asks whether to save the definition of Arch recorded so far. When we confirm, the "after" picture of the icon representing the Arch operation is filled in with a miniature picture of the final state of the screen. The newly defined operation is represented by a domino of before and after pictures of the example presented by the user to define the command.

Figure 10: Final version of the Arch icon



Now, we can use the Arch operation just like any other of Mondrian's operations. Here are some examples of applying the Arch operation to other rectangles. Slight alignment inaccuracies that appeared in the original are removed, and the thickness of the arch elements is made proportional to what it was in the original.

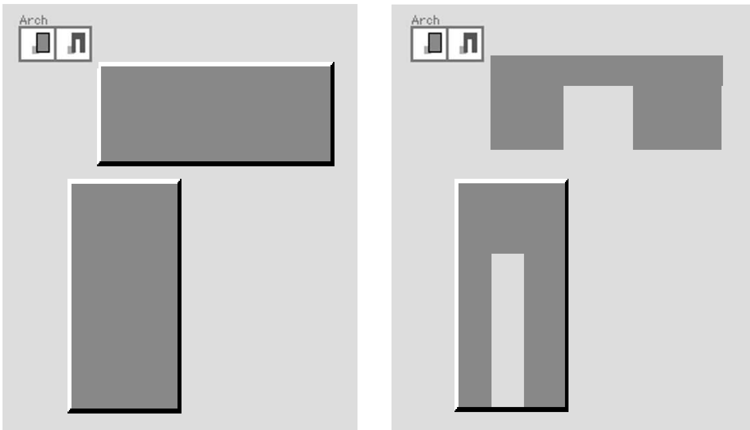


Figure 11: Before and after applying the Arch operation

Storyboards: A Pictorial Representation Of Program Code

A problem in many previous programming by demonstration systems has been how to give the user feedback about what the system has learned as a result of recording operations. The programming by demonstration system generates code in a programming language, but the user of the graphical interface does not want to see this code as the definition of the new command. The code is confusing because the user did not write that code him or herself.

We borrow the idea of a *storyboard* from animation and multimedia design. Storyboards are graphs of snapshots of the state of a moving image, with time along the horizontal axis. Storyboards may be one-dimensional, or one-and-half dimensional, with the half dimension being discrete "tracks". Events appearing vertically aligned in different tracks are synchronized in time. Storyboards are effective because they provide a static view of a dynamic process, and help the user visualize how events unfold over time.

The execution of a program is, like an animation, a sequence of events that unfold over time. If the events are interactions with a graphical interface, this suggests that a storyboard can be an effective means of visualizing program states. What's really important about program text in a conventional programming language is that it provides a static description of the dynamic process of executing the program. Storyboards can also provide this static view, but in a pictorial rather than textual way.

Mondrian's storyboards are sequences of miniature snapshots of the state of the screen. Each snapshot represents the state of the screen just before invocation of a command. These storyboards can be thought of as "expansions" of the before and after domino icons to include intermediate states. The storyboard is displayed by shift-clicking on the icon. The storyboard consists entirely of images that the user has seen before in the course of interaction, so that each image serves as a visual reminder to the user of his or her intent at that point in time.Each frame of the storyboard is labeled with the name of the operation invoked and a miniature version of its icon. This is the way of saying that the snapshot "stands for" the use of that operation in that context.

When multiple-example capabilities such as those found in Tinker [see the chapter on Tinker] are installed into Mondrian, the storyboard will be composed of multiple tracks, one for each presented example. Conditionals require branching that makes a totally linear storyboard inappropriate. We intend to enhance the storyboard interface with most of the operations appropriate for browsing and editing program code: hierarchical level of detail control, editor, reversible stepper, tracer, etc.

Figure 12: Storyboard for the Arch

Storyboard representations for programming appear in [Fineblum and Lieberman 91] and for graphical editing in [Kurlander and Feiner 90]. Animated icons, such as "micons" [Brondmo and Davenport 90] and those described in [Baecker, Small, and Mander 91] are a kind of dynamic storyboard.

Speech Output: User Feedback About Generalization

Some kind of feedback should be given to the user about how the system is interpreting the user's actions. Ideally, this feedback might be given in real time as the user is performing the action or soon afterward, so that the user may easily correct the misinterpretations. However, we do not want the feedback to be disruptive and draw the user's attention away from the drawing task. If we were to pop up a window and print out line-by-line feedback, the user's attention would be distracted from the drawing surface.

The speech channel is perfect for providing feedback which does not interfere with visual action. Mondrian uses speech synthesis software to provide a running commentary about the system's interpretation of the user's actions. Mondrian has a very simple natural language generator that "reads aloud" the code generated by the system. It strings together complete sentences from templates associated with the generated abstractions. An example of Mondrian's verbal description is given below

The template for the arch is simply referred to as "the first argument". The system generates names for objects introduced in the course of the interaction. We will also allow user-supplied names, which will then be used in the system's commentary.

Start the definition of Arch.
Draw a rectangle from the left top corner of the first argument
to a point on the first argument, and call it Rectangle 3.
Draw a rectangle from the left top corner of Rectangle 3 to a point
on the first argument, and call it Rectangle 5.
Draw a rectangle from the right top corner of Rectangle 5 to a point
on the first argument, and call it Rectangle 7.
Delete the first argument.
Make a group out of Rectangle 3 and Rectangle 5 and Rectangle 7.
End the definition of Arch.

Figure 13: Mondrian's narration of the Arch procedure

Another approach we are considering is to use sampled speech, using predefined samples for the "canned" portions of the text, and digitizing the user's pronunciation of names for variables during the interaction. This would be more intelligible than current low-quality voice synthesizers.

What Code Did Mondrian Generate?

Though the external representation presented to the user is a pictorial storyboard, the internal representation of the generalized program is a Lisp function. Below is the Lisp code generated by Mondrian for the Arch example.

The function ARCH takes two arguments, the object representing the graphical editor [INTERACTOR] and a list of arguments [SELECTION]. It consists of five function calls, each to an action routine corresponding to a single interactive command, three RECTANGLEs, a DELETE and a GROUP. Looking at the first rectangle as a typical case, its left top corner is the left top corner of the first argument [(LEFT-TOP (NTH 0 SELECTION))] and its right bottom corner is a point on the first argument that is a small fraction of the way across and all the way down to the bottom.

Figure 14: The Lisp code for Arch

```
(DEFUN ARCH (INTERACTOR SELECTION)
  (RECTANGLE-COMMAND (LEFT-TOP (NTH 0 SELECTION))
                     (POINT-ON-OBJECT (NTH 0 SELECTION) 12/95 1)
                     INTERACTOR
                     'RECTANGLE-5)
  (RECTANGLE-COMMAND (LEFT-TOP (FIND-OBJECT-NAMED INTERACTOR 'RECTANGLE-5))
                     (POINT-ON-OBJECT (NTH 0 SELECTION) 93/95 8/51)
                     INTERACTOR
                     'RECTANGLE-6)
  (RECTANGLE-COMMAND (POINT-ON-OBJECT (FIND-OBJECT-NAMED  INTERACTOR
                                       'RECTANGLE-6)
                                       76/91
                                       1/14)
                     (RIGHT-BOTTOM (NTH 0 SELECTION))
                     INTERACTOR
                     'RECTANGLE-7)
  (REMOVE-OBJECT INTERACTOR (NTH 0 SELECTION))
  (GROUP-COMMAND (LIST (FIND-OBJECT-NAMED INTERACTOR 'RECTANGLE-5)
                       (FIND-OBJECT-NAMED INTERACTOR 'RECTANGLE-6)
                       (FIND-OBJECT-NAMED INTERACTOR 'RECTANGLE-7))
                 INTERACTOR
                 'GROUP-8))
```

The last argument to commands that generate new objects is a name for the new object. This name is used to refer to the object when subsequently selected as an argument to another command, as the three new rectangles are selected as arguments to the final GROUP command.

The code produced by Mondrian is not too different, except for form, from what would plausibly be produced by hand-coding. Specifically, no absolute screen coordinates or other constants appear solely as accidental artifacts of the interaction. The only numbers that appear are ratios that indicate the proportions of the arch components.

Generalization Heuristics: Dependencies

By inscribing rectangles in the input template to the Arch operation, we are indicating to Mondrian a dependency of the dimensions of the newly drawn rectangles on the input template. One way that Mondrian interprets the user's actions is by searching for a significant relationship between the characteristics of the objects drawn and the input arguments. Mondrian presumes that the input arguments were chosen because of their salience for the desired generalization, so that relationships involving argument objects are given priority.

The choice of the significant relationships depends on the nature of the graphical objects. In the case of Mondrian's rectangles, we recognize relationships such as LEFT, RIGHT, TOP, BOTTOM, CENTER, ABOVE, BELOW. How a specific user action is interpreted depends on the kind of input expected by the user interface at a given moment. A single mouse click might be interpreted as indicating a point, one of the visible rectangle objects, an invocation of a command, etc. depending upon the context.

For points, coincidence with special points such as corners and centers is noted. A point on one of the visible rectangle objects but not at one of the special locations is noted by its relative position on that object. Objects that are input arguments to the procedure being defined are significant, and other objects are represented by their relationship, if any, to the argument objects. Focusing on the argument objects helps prevent accidental matches that might otherwise occur. Points or other objects that otherwise have no special relations are noted by an absolute reference; their name, if they possess one, or their coordinates. Objects referred to by name are the equivalent of global variables in conventional languages.

The system has a default set of heuristics for prioritizing recognition of these relations. These heuristics are normally fixed [though they are described internally by an object-oriented protocol and could be easily extended by a programming user] so that the user does not have to concern his or herself with disambiguating underconstrained relations while the program is being defined. We want to keep the teaching interaction as rapid as possible, so we do not encumber the interaction with queries to disambiguate input. We envision that the user who wishes more control will supply advice to the system with a separate *generalization editor* interface, that will allow interactive editing of the generalization heuristics.

Related Work

Closest to this work is Chimera [see the Chimera chapter in this volume]. Chimera also records user actions in a graphical editor and generalizes them to define new operations. Chimera also displays sequences of actions to the user in the form of storyboards, with detail-suppression and editing techniques not found in Mondrian. Storyboards were originally introduced in [Kurlander and Feiner 88b] for recording action sequences for undo and redo operations.

Mondrian’s use of dominoes as a static visual representation of an operation, and the relation between dominoes and storyboards, is significant. The importance of dominoes is that they fold the newly defined operation back into the user interface in the same iconic form as the already-existing operations. The new operation can then be recorded as part of another procedure defined by demonstration and everything appears in a consistent visual language. The use of synthesized speech for feedback about the system’s interpretation of user actions is also unique to Mondrian.

Mondrian's generalization has some differences from Chimera's. The most obvious difference is the order in which generalization advice is given -- in Mondrian at the start of the demonstration, in Chimera afterwards. It was done this way in Mondrian for several reasons: so speech could report the generalizations; so generalization could be shown in the dominoes; to avoid a dialog box asking the user how to generalize clicks and drags [see also Turransky’s chapter on voice input for an alternative] and to facilitate adding Tinker's multiple-example capability for defining recursive functions [this issue is beyond the scope of the present paper, but see the Tinker chapter]. Each style might be better in some circumstances or for some users. Finally, the representation of the result of generalization is different in Mondrian than in Chimera. Mondrian creates a Lisp program, whereas Chimera does not have an independent procedural representation of the results of generalization.

David Maulsby's Metamouse is also a graphical editor that can learn new procedures through programming by demonstration. Mondrian differs in its function-and-argument structure for graphical operations. New operations explicitly become available as iconic operations parameterized by their arguments, whereas Metamouse learns only a single global procedure. Metamouse also lacks any static description of the resulting procedure visible to the user, such as Mondrian's storyboards.

Allen Cypher's Eager is a programming by demonstration interface agent for Hypercard that looks for repetitive operations and proposes them as candidates for generalization. Eager also lacks a function and argument model, and a visible static representation. Brad Myers' Peridot is a by-demonstration interface editor driven by a rule-based recognition procedure. Peridot differs fundamentally from Mondrian, Metamouse and Chimera in that it generalizes from states of the interface rather than recorded actions.

Mondrian is, well, less "eager" than Eager, Peridot and Metamouse in that it does not "jump to conclusions" about the intent of repetitive operations. Mondrian's instructible interface metaphor relies on the user to explicitly indicate where repetition is taking place. A repetitive operation must be indicated by clicking on the icon representing the action currently being defined. This will be especially important in the definition of recursive functions, where functions are only partially defined at any moment, and repetition may or may not indicate recursive invocation. There is a tradeoff between an aggressive generalization policy, which is more automatic in the cases where it is able to correctly recognize a pattern of actions, and a more conservative generalization policy that affords greater user control and flexibility.

The author's earlier Tinker system [see the Tinker chapter] was a programming by demonstration system that had the capability of incorporating multiple examples to define conditional and recursive procedures. We intend to bring this capability into Mondrian in the near future. Tinker was one of the most general programming by demonstration systems, having the potential of producing any program expressible in Lisp.

Several other programming by demonstration systems were highly influential to me, including Laura Gould and Bill Finzer's Programming by Rehearsal and Dan Halbert's Smallstar The landmark system that introduced the techniques of modern programming by demonstration and visual programming systems was David Canfield Smith's Pygmalion. Eager, Peridot, Metamouse, Tinker, Programming by Rehearsal, Smallstar and Pygmalion are all described in chapters of this book

Acknowledgments

The Visible Language Workshop at the MIT Media Laboratory is supported by research grants from Alenia, Apple, DARPA, Kansa, NYNEX, Paws, and Digital.