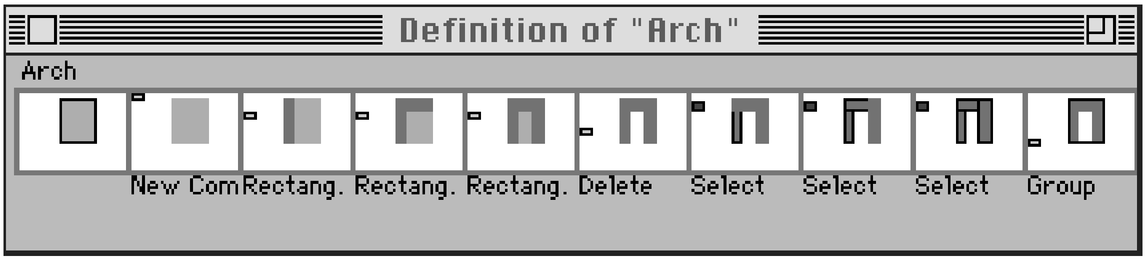
Dominoes and Storyboards: Beyond "Icons on Strings"

Henry Lieberman

Visible Language Workshop
Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. USA



ABSTRACT

Practically since graphic displays were first hooked to computers, the idea of representing computer programs by pictures has attracted researchers. However, to date, most proposals for visual programming languages have adhered to a set pattern: fixed pictures symbolizing program components, connected by lines or arrows symbolizing relationships between the program components. This "icons on strings" approach, while it can be useful, is not the only way of visualizing programs.

In this paper, I explore one alternative: representing a program through visual examples of the state of its execution. I present two related techniques: *dominoes*, which replace the traditional icons as representations of operations; and *storyboards*, which replace iconic circuitry as the representation of program code. These have been implemented in Mondrian, a graphic editor extensible through programming by example.

Most proposed visual programming languages are "icons on strings"

The dream of visual programming has been to replace textual programming languages as the medium through which computers are instructed with a language of pictures. As with natural languages, there is no one visual language that is privileged: many provide sufficient expressive power, but emphasize different kinds of relations.

Despite the apparent variety of proposed visual languages, most fall into a category I call *icons on strings*. These languages are characterized by

- *Iconic pictures*, usually static, that have a one-to-one mapping to the concepts that make up the program, contained on the screen in a box of bounded size, and
- *Connections between them*, usually represented visually as lines or arrows. The connections represent relationships between the concepts.

Languages may differ in what relations they choose to represent. The illustration on the top of the next page, a mosaic of several systems from the past two years' Visual Languages conferences, shows some of the many icon-on-string languages.

The only major variant of the icon-on-string idea has been to replace the "strings" as a means of connecting the icons with the relation of containment. Icons can be contained inside one another to express relationships like the function-argument relation.This might be called "icons in icons". Containment relations are limited to trees, whereas lines can express full graphs. Boxer [Abelson and diSessa] and VennLisp [Lakin 86] are examples of containment tree languages. I used the idea of containment in three dimensions in [Lieberman 89].

What's wrong with "icons on strings"?

Like program code in textual languages, graph representations of code visually expose only the representations of program code and relations between program modules. However, many tasks in programming require understanding not only the relationship between program components, but the relationship between program components and the data being operated upon.

This is especially true in debugging. Debugging is a visualization task that requires discovering which parts of a program are responsible for the production of incorrect data, or which program components need to be added to modified to produce desired objects which do not appear in the incorrect program. For this, it is necessary to present visually not just the code, or just the data, but the relations between code and data.

Algorithm animation is a valuable debugging tool, because it can dynamically represent the temporal evolution of data in a program. However, without representation of the code itself and indications of the connections between code and data, the task of associating undesired behavior with the characteristics of the algorithm that produced it is left to the unassisted thought processes of the programmer.

Another problem with graph representations is that they scale up poorly. The typical symptom is that in highly interconnected graphs, the density of connections reduces the visual impression to a "plate of spaghetti". Abstraction mechanisms, such as zoom and pan, level of detail control, link consolidation, browsers, and other interactive techniques have as yet been insufficiently explored to alleviate this concern.

Concrete visual examples illustrate abstract operations

An alternative to the icons-on-strings metaphor is to use concrete *visual examples* as the representation. In their simplest form, the examples are simply visual snapshots of the state of the screen in the user interface, usually reduced in size.

Concretely, the screen snapshots represent the state of execution of a program in one particular situation. However, more abstractly, the visual examples may represent the operations which gave rise to them. This becomes much more effective when more than one example is used. Human visual recognition procedures are attuned toward easily recognizing the differences between similar scenes, or imagining animation of a series of successive frames illustrating a dynamic process.

Visual examples solve the code vs. data problem, because the visual example explicitly represents a typical example of the data being operated upon. The code is implicitly represented by a visual state transformation that occurs between two or more visual examples. Since the association between code and data is always visually apparent, incorrect data helps to visually locate the code responsible.

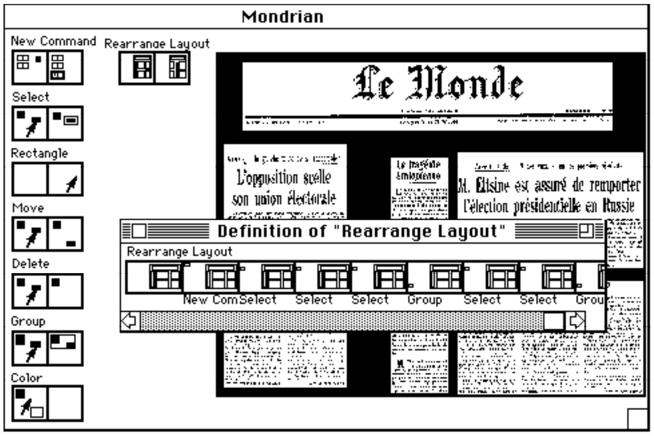
Visual examples need not be literal copies of the screen. They may be modified in a variety of ways in order to better suggest what they represent. Elements of the interface deemed unimportant may be omitted, while important ones emphasized using color or other highlighting. They may be annotated with additional labeling that directs the user's attention to salient features.

Mondrian: A teachable graphical editor that uses a visual language of examples

Mondrian [Lieberman 92] is an object-oriented graphical editor, similar to MacDraw, that can learn new graphical procedures through programming by example. The user demonstrates the steps of a graphical procedure through interaction with the system, using a concrete visual example. The system remembers the steps and can generalize them to produce a program that will perform an "analogous" procedure on future examples.

Mondrian's programming-by-example approach, details of the generalization and other aspects are covered in a companion paper [Lieberman 92]. This paper emphasizes the visual language aspects. Mondrian uses visual examples as the sole visual representation of operations, where most direct-manipulation graphical editors use icons or menu selections. Built-in operations use visual examples to teach the user about how they work. A unique aspect of Mondrian's interface is that the system automatically constructs iconic representations from visual examples to represent user-defined functions.

Here, a picture of a typical Mondrian screen. Built-in functions are in the leftmost column, and a user-defined function Rearrange Layout captures a page-layout manipulation demonstrated on the example layout from *Le Monde* by the user.



Mondrian

Visual examples are the language of visual problem solvers

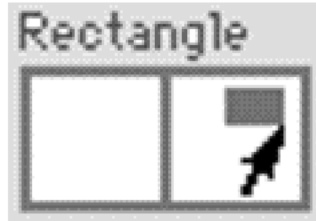
Expert visual problem solvers, such as graphic designers, architects, industrial designers, multimedia designers and animators use visual examples as their primary means of communication. Rarely do they resort to directed-graph diagrams to express their ideas to colleagues, or to teach beginning students. Directed-graph representations are used more often by people who are expressing organizational charts, flowcharts, or block diagrams of networks, rather than by people whose subject matter includes visual communication.

A quick look at any book intended to teach graphic design or any visual design discipline will confirm the central role of visual examples. [Arent, Vertelney, and Lieberman 89] discusses the differences in computer science and visual design viewpoints.

Dominoes: Icons that explain the operation of the functions they represent

Mondrian's basic visual representation of a function is called a *domino*. A domino looks superficially like an icon in conventional direct-manipulation systems.

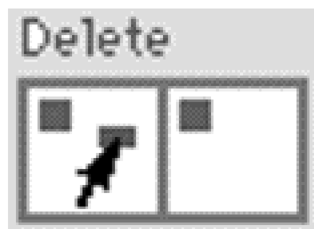
The idea of a domino is that it portrays an example of the use of the command that it represents. The domino is composed of two parts: the *before* and *after* pictures. The before picture represents the precondition of the operation, the after picture the postcondition. The domino represents the function that performs the transition from the state shown in the before picture, to the state shown in the after picture.



Rectangle domino

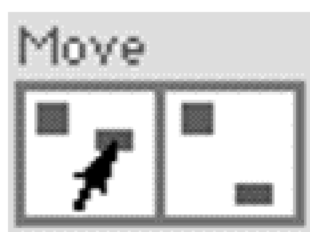
For example, for the Rectangle function, the before state is a blank screen. The after state is a screen which contains a rectangle. Notice that the cursor is represented in the after picture, to suggest the rubber-band dragging in the specification of the rectangle's size and position. However, notice that the drawing is "not to scale"; as cursors are rarely half the size of the screen!

The Delete domino displays a red and a blue object in the before picture, with the blue object selected. In the after picture, it is gone.



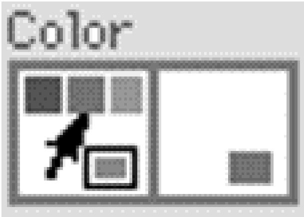
Delete domino

The object which is not selected in the before panel of the Delete domino, while not directly involved in the operation, is informative in the visual representation because it provides a contrast to the transformation, and thus helps to establish context. The Move domino is similar.



Move domino

One key difference between the domino and the conventional icon is that the icon is a static picture. The domino can also carry state information about how the command will behave, which can be changed dynamically.



Color domino

The Color command is used to select the default color for drawing. When a color is selected, two dominoes change state in response. The color domino displays several color choices and illustrates that the color chosen in the before picture will be applied to the selected object in the after picture. The chosen color which appears in the Color domino will be changed to match the current default color of the graphic editor. The color of the Rectangle domino will also change to reflect the fact that rectangles drawn will now appear in the newly selected color. Showing the anticipated effect of a command to help the user decide in selecting it could be termed *graphical futures*, by analogy with Kurlander and Feiner's graphical histories.

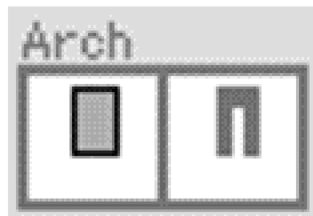
The New Command domino is perhaps the only one that has no analog in commercial graphical editors. It enables Mondrian's learning capability.

New Command domino

The before picture shows a column of dominoes, representing the column of dominoes that Mondrian displays upon startup. In the after picture, the column of dominoes has an additional member, denoting the new domino that the New Command operation is going to introduce.

A further subtlety is that the before and after pictures in the newly created domino that appears in the after picture of the New Command domino are the before and after pictures of the New Command domino itself [got that?]. The idea of the New Command icon is that it takes the state of the screen at the time it is invoked, and uses that as the input state for a new operation to be defined. Graphical objects selected at the time of invocation of New Command are assumed to be arguments to the operation being defined, in the order in which they are selected. The user may then perform graphical operations that are to be remembered and generalized on the input arguments, so that the procedure may later be applied to a different set of arguments. The initial and final states of the example demonstration are captured to become the before and after pictures of the newly defined operation.

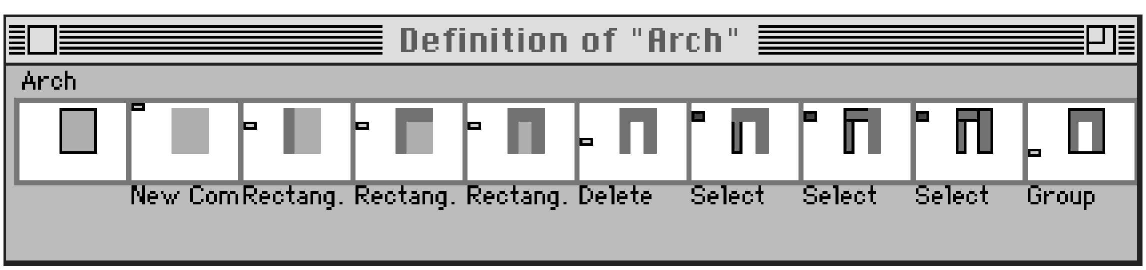
Here is a system-constructed domino that represents a procedure that transforms a rectangle into an Arch.

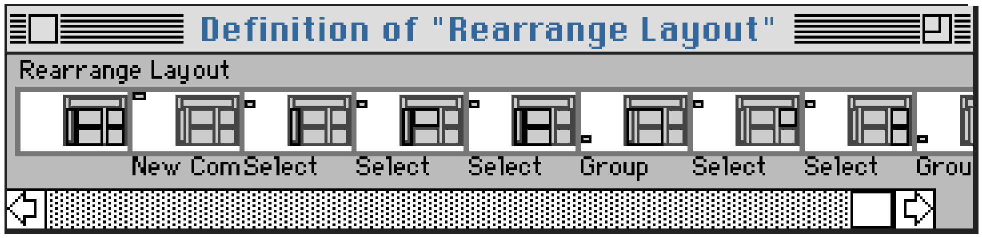


Generated Arch domino

The user started out by demonstrating the procedure on a rectangle that appears in the before panel. The resulting screen at the end of the definition of the Arch command has replaced the rectangle by an arch inscribed in it.

New users are sometimes surprised to see that the screens captured by the New Command operation also include objects that were not directly involved in the demonstrated procedure. Whether to do this or not is a design tradeoff in the visual language. An advantage of retaining non-argument objects is that they provide a visual context which contrasts with the transformation of the argument objects, much as was seen in including the stationary object in the illustration of the Move operation. Mondrian also has the capability to define procedures which use extraneous objects as "global variables" in the program. Since the system can't know in advance which





objects may be referenced globally, it simply retains them all. It is also easy to change the system's behavior so that objects not directly involved in the definition of a procedure do not appear in the domino for that procedure.

Storyboards: Visual examples as program code

A problem in most prior programming by example systems has been how to show the user feedback as to what the system has learned by the interaction. This feedback will be particularly important for debugging and editing. The system usually generates code in a programming language, but showing textual programs to a user who has presented an example through direct manipulation is problematic.

Mondrian's solution is a use a visual *storyboard* as the representation of code. Shift-clicking on the icon for the user-defined operation will bring up a visual storyboard consisting of a "cartoon" of the successive states of the demonstrated example, labeled with the operation chosen at each step.

Above, a storyboard for the definition of an Arch program. The Arch is inscribed in a background rectangle, which is deleted after it has served its purpose of constraining the size and position of the arch.

The storyboard really *is* the code for the program. A conventional programming language, like Lisp, is composed merely of functions and arguments to these functions. The storyboard contains each panel annotated with the function which has been called. The arguments are graphical objects displayed as selected in the storyboard panel. Thus the full function-and-argument structure of the resulting program is recoverable from the storyboard representation. As an additional representation of the operation chosen, a miniature version of the selected domino is displayed in each panel.

What's important about program code is not that it is a document written in a conventional textual programming language like Lisp or C. Code, like music notation, provides a static representation that helps a person visualize how events unfold over time. It is a way of using space to represent time.

You can think of the storyboard, not as a sequence of individual panels, but as a sequence of overlapping dominoes. The operation named by the text caption is represented by the panel above the caption, *and* the panel immediately preceding it.

For example, in the storyboard for the Arch procedure, there are three different dominoes that represent the same Rectangle operation. These variants, together with the Rectangle domino which appears in the initial column of built-in operations, illustrate the fact that the mapping of pictures to operations is many-to-one, unlike conventional iconic languages.

In the storyboard for a newspaper layout procedure, the storyboard panels are not simple copies of the screen. Individual articles would not be legible at the screen resolution of the domino. Each panel represents a sketch of the configuration of articles on the page, as the position and size are the salient aspects.

This illustrates that an abstraction step may be applied to the screen state in the storyboard creation process. We will experiment with further annotating the storyboard with the generalization applied during the inference process.

We intend to develop the storyboard representation as a full-fledged programming environment, including tools such as editors and steppers. A stepper will have the resulting of animating the replay of the storyboard, with user control over speed and detail.

Tell me a story: Speech synthesis as an audio storyboard

Mondrian also uses synthesized speech to present an "audio storyboard". A simple natural language generator transforms Lisp expressions into natural language sentences. This "reads the code" to the user. Below, a segment of Mondrian's explanation of the Arch procedure.

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.

Speech transcription

In the current version, the speech feedback is more useful as simultaneous output when the user is drawing than when the user is reviewing the storyboard. Use of the speech channel is good because the user's hand-eye coordination is occupied with the drawing task. Popping up additional windows for feedback in this situation would be disruptive.

Related work

The most closely related work to this is [Kurlander and Feiner 92], where a storyboard is used for defining macros from graphical histories [Kurlander and Feiner 90]. Unique to the present work is the use of the dominoes to fold the procedure defined by the storyboard back into the interface in a uniform visual manner, and the audio storyboard. Unique to their work are abstraction techniques such as coalescing successive frames, focusing on relevant portions of the screen, and the editing techniques. These techniques could profitably be incorporated into Mondrian's storyboards.

Michelle Fineblum and the author built a system which used a multitrack storyboard to represent a stack trace used in debugging a natural language program [Fineblum and Lieberman 91]. The examples were textual, but color-coding identified function structure and other properties of the state of the program. The idea of animating icons is also developed in [Baecker and Small 90] and [Brondmo and Davenport 90]. A domino can be thought of as a "two-frame" animated icon.

Conclusion

Visual examples can serve as a powerful way of breaking out of the "icons on strings" stereotype of visual programming. We have explored three example-based techniques : dominoes, dynamic icons that represent an operation through before-and-after pictures; storyboards, which represent a sequence of operations through overlapping dominoes; and an audio storyboard using natural language output.

In traditional program-graph or data-flow representations, fixed iconic pictures represent program operations, and the data on which they operate is implicit. With visual example-based representations, examples of the data are explicit, and the operation is only implicitly represented by transitions between example states. Since people are by nature good at generalizing from examples, visual examples can serve as a congenial visual programming tool, especially for visually-oriented users.

Acknowledgments

The Visible Language Workshop is supported by research grants from Alenia, Apple Computer, DARPA, Digital Equipment Corp., Kansa Insurance Co., NYNEX, and Paws, Inc.

References

[Abelson and diSessa 87] Hal Abelson and Andy di Sessa, Boxer: A Principled Design for a Graphical Programming Environment, Communications of the ACM, 1987.

[Arent, Vertelney, Lieberman 89] Michael Arent, Laurie Vertelney and Henry Lieberman, Two Disciplines in Search of an Interface: Reflections on a Design Problem, in The Art of Human-Computer Interface Design, Brenda Laurel, ed., Addison-Wesley.

[Brondmo and Davenport 90] Hans-Peter Brondmo and Glorianna Davenport, Creating and Viewing The Elastic Charles, a Hypermedia Journal, in Hypertext: State of the Art, McAleese and Green, eds.., Intellect, 1990.

[Baecker, Small, Mander 91]Ron Baecker, Ian Small and Richard Mander, Bringing Icons to Life, CHI 91.

[Chang 90] S.K. Chang, ed.,Principles of Visual Programming Systems, Prentice-Hall.

[Cypher 91] Allen Cypher, Eager: Programming Repetitive Tasks by Example, CHI 91.

[Cypher, et. al. 92] Allen Cypher, Dan Halbert, David Kurlander, Henry Lieberman, Brad Myers, eds., Watch What I Do: Programming by Demonstration, to appear, 1992.

[Kurlander and Feiner 90] David Kurlander and Steven Feiner, A Visual Language for Browsing, Undoing and Redoing Graphical Interface Commands. In Visual Languages and Visual Programming, S.K. Chang, ed., Plenum Press.

[Lakin 86] Fred Lakin, Spatial Parsing for Visual Languages, In Visual Programming, S.K. Chang, T. Ichikawa and P. Ligomenedes, ed., Plenum Press.

[Lieberman 87] Henry Lieberman, An Example Oriented Environment for Beginning Programmers, in Artificial Intelligence and Education, R. Lawler and M. Yazdani, eds., Ellis Horwood, Chichester, England.

[Lieberman 89] Henry Lieberman, A Three-Dimensional Representation for Program Execution, IEEE Workshop on Visual Languages, Rome, Italy, 1989.

[Lieberman and Fineblum 91] Henry Lieberman and Michelle Fineblum, Storyboard-Based Programming Tools, CHI 91 (short paper).

[Lieberman 92] Henry Lieberman, Mondrian: A Teachable Graphical Editor, in [Cypher, et. al. 92]

[Shu 88] Nan Shu, Visual Programming, Van Nostrand Reinhold.

[Smith 76] David Canfield Smith, Pygmalion: A Creative Programming Environment, Birkhauser-Verlag.