In: Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), John Wiley & Sons Inc., New York, (to appear approx. 1999).

Visual Programming

Margaret M. Burnett, Oregon State University

Visual programming is programming in which more than one dimension is used to convey semantics. Examples of such additional dimensions are the use of multi-dimensional objects, the use of spatial relationships, or the use of the time dimension to specify "before-after" semantic relationships. Each potentially-significant multi-dimensional object or relationship is a token (just as in traditional textual programming languages each word is a token) and the collection of one or more such tokens is a *visual expression*. Examples of visual expressions used in visual programming include diagrams, free-hand sketches, icons, or demonstrations of actions performed by graphical objects. When a programming language's (semantically-significant) syntax includes visual expressions, the programming language is a *visual programming language* (VPL).

Although traditional textual programming languages often incorporate two-dimensional syntax devices in a limited way--an x-dimension to convey a legal linear string in the language, and a y-dimension allowing optional line spacing as a documentation device or for limited semantics (such as "continued from previous line")--only one of these dimensions conveys semantics, and the second dimension has been limited to a teletype notion of spatial relationships so as to be expressible in a one-dimensional string grammar. Thus, multidimensionality is the essential difference between VPLs and strictly textual languages.

When visual expressions are used in a programming environment as an editing shortcut to generate code that may or may not have a different syntax from that used to edit in the code, the environment is called a *visual programming environment* (VPE). Visual programming environments for traditional textual languages provide a middle ground between VPLs and the widely-known textual languages. In contrast to just a few years ago, when strictly textual, command-line programming environments were the norm, today VPEs for traditional textual languages are the predominant kind of commercial programming environment. Commercial VPEs for traditional languages are aimed at professional programmers; these programmers use the textual languages they already know, but are supported by the graphical user interface techniques and accessibility to information that visual approaches can add. VPEs for traditional languages serve as a conduit for transferring VPL research advances into practice by applying these new ideas to traditional languages already familiar to programmers, thus affording a gradual migration from textual programming techniques to more visual ones. VPLs are usually integrated in their own custom environments, so for the remainder of this article, we will follow convention, using the term VPEs to mean VPEs for traditional languages.

History

The earliest work in visual programming was in two directions: visual approaches to traditional programming languages (such as executable flowcharts), and new visual approaches to programming that deviated significantly from traditional approaches (such as programming by demonstrating the desired actions on the screen). Many of these early systems had advantages that seemed exciting and intuitive when demonstrated with "toy" programs, but ran into difficult problems when attempts were made to extend them to more realistically-sized programs. These problems led to an early disenchantment with visual programming, causing many to believe that visual programming was inherently unsuited to "real" work--that it was just an academic exercise.

To overcome these problems, visual programming researchers began to develop ways to use visual programming for only selected parts of software development, thereby increasing the *number* of projects in which visual programming could help. In this approach, straightforward visual techniques were widely incorporated into programming environments that support textual programming languages, to replace cumbersome textual specification of GUI layout, to support electronic forms of software engineering diagrams for creating and/or visualizing relationships among data structures, and to visually combine textually-programmed units to build new programs. Successful commercial VPEs soon followed; among the early examples were Microsoft's Visual Basic (for Basic) and ParcPlace Systems' VisualWorks (for Smalltalk). Another group of commercial VPEs, focused primarily on large-grained programming, are the Computer-Aided Software Engineering (CASE) tools that support visual specification (for example, using diagrams) of relationships among program modules, culminating in automatic code generation of composition code.

Other visual programming researchers took a different approach--they worked to increase the *kinds* of projects suitable for visual programming through the development of domain-specific visual programming systems. Under this strategy, the addition of each new supported domain increased the number of projects that could be programming visually. An added benefit that followed was improved accessibility--end-users were sometimes able to use these new systems. The developers of domain-specific VPLs and VPEs found that providing ways to write programs for one particular problem domain eliminated many of the disadvantages found in the earliest approaches, because they supported working directly in the communication style of the particular problem domain--using visual artifacts (e.g., icons and menus) reflecting the particular needs, problem-solving diagrams, and vocabulary specific to that domain--and never forced users to abandon that communication style. This approach quickly produced a number of successes both in research and in the marketplace. Today there are commercial VPLs and VPEs available in many domains; examples include programming laboratory data acquisition (National Instruments' LabVIEW), programming scientific visualizations (Advanced Visual Systems' AVS), programming telephone and voice-mail behavior (Cypress Research's PhonePro), and programming graphical simulations and games (Stagecoach Software's Cocoa). A number of software-agent generators are starting to become embedded in personal computing software as well, allowing macros that assist with repetitive tasks to be inferred from end-user manipulations (as in Chimera, for example, which is discussed in the next section).

The original challenge--to devise VPLs with enough power and generality to address an ever-expanding variety of programming problems--is an ongoing area of research. One goal of this research is to continue to improve the ways visual programming can be used. Another goal is to provide the same kinds of improvements in general software development as are already available for programming in some domain-specific areas. But although this work is still primarily in the research stage, commercial VPLs with the characteristics needed for general-purpose programming have emerged and are being used to produce commercial software packages; one example is Pictorius International's Prograph CPX.

Strategies in Visual Programming

Because VPEs employ visual ways of communicating about programs, the visual communication devices employed by a VPE can be viewed as a (limited) VPL. Hence, the strategies used by VPEs are a subset of those possible for VPLs. Because of this subset relationship, much of the remaining discussion of visual programming will focus primarily on VPLs.

VPL Strategies

A common misunderstanding is that the goal of visual programming research in general and VPLs in particular is to eliminate text. This is a fallacy--in fact, most VPLs include text to at least some extent, in a multidimensional context. Rather, the overall goal of VPLs is to strive for improvements in programming language design. The opportunity to achieve this comes from the simple fact that VPLs have fewer syntactic restrictions on the way a program can be expressed (by the computer or by the human), and this affords a freedom to explore programming mechanisms that have not previously been tried because they have not been possible in the past.

The most common specific goals sought with VPL research have been (1) to make programming more accessible to some particular audience, (2) to improve the correctness with which people perform programming tasks, and/or (3) to improve the speed with which people perform programming tasks.

To achieve these goals, there are four common strategies used in VPLs:

Concreteness: Concreteness is the opposite of abstractness, and means expressing some aspect of a program using particular instances. One example is allowing a programmer to specify some aspect of semantics on a specific object or value, and another example is having the system automatically display the effects of some portion of a program on a specific object or value.

Directness: Directness in the context of direct manipulation is usually described as "the feeling that one is directly manipulating the object" [19]. From a cognitive perspective, directness in computing means a small distance between a goal and the actions required of the user to achieve the goal [11, 13, 17]. Given concreteness in a VPL, an example of directness would be allowing the programmer to manipulate a specific object or value directly to specify semantics rather than describing these semantics textually.

Explicitness: Some aspect of semantics is explicit in the environment if it is directly stated (textually or visually), without the requirement that the programmer infer it. An example of explicitness in a VPL would be for the system to explicitly depict dataflow relationships (program slice information) by drawing directed edges among related variables.

Immediate Visual Feedback: In the context of visual programming, immediate visual feedback refers to automatic display of effects of program edits. Tanimoto has coined the term *liveness*, which categorizes the immediacy of semantic feedback that is automatically provided during the process of editing a program [21]. Tanimoto described four levels of liveness. At level 1 no semantics are implied to the computer, and hence no feedback about a program is provided to the programmer. An example of level 1 is an entity-relationship diagram for documentation. At level 2 the programmer can obtain semantic feedback about a portion of a program, but it is not provided automatically. Compilers support level 2 liveness minimally, and interpreters do more so because they are not restricted to final output values. At level 3, incremental semantic feedback is automatically provided whenever the programmer performs an incremental program edit, and all affected onscreen values are automatically redisplayed. This ensures the consistency of display state and system state if the only trigger for system state changes is programmer editing. The automatic recalculation feature of spreadsheets supports level 3 liveness. At level 4, the system responds to program edits as in level 3, and to other events as well such as system clock ticks and mouse clicks over time, ensuring that all data on display accurately reflects the current state of the system as computations continue to evolve.

VPL Examples

In this section, we discuss four example VPLs to demonstrate several ways in which the strategies of the previous section have been employed.

Imperative Visual Programming by Demonstration

Chimera [14] is an example of the most common way imperative programming is supported in

VPLs, namely by having the programmer demonstrate the desired actions. In the case of Chimera, the "programmer" is an end user: hence, Chimera an example of a VPL aimed at improving accessibility of programming certain kinds of tasks.

The domain of Chimera is graphical editing. As an end user works on a graphical scene, he or she may find that repetitive editing tasks arise, and can indicate that a sequence of manipulations just performed on a scene should be generalized and treated as a macro. This is possible because the history of the user's actions is depicted using a comic strip metaphor (see Figure 1), and the user can select panels from the history, indicate which of the objects should be viewed as example "parameters," (graphically) edit the actions depicted in any of the panels if desired, and finally save the sequence of edited panels as a macro. Chimera uses inference in determining the generalized version of the macro; use of inference is common in by-demonstration languages, and its success depends on limited problem domains such as Chimera's. However, there are also a number of by-demonstration languages that do not use inference, one example of which is Cocoa (discussed later in this article).

Chimera is at liveness level 3; that is, it provides immediate visual feedback about the effects of program edits. Since these effects are rendered in terms of their effects on the actual objects in the program, this is an example of concreteness. Directness in Chimera is used in that the way program semantics are specified is by directly manipulating objects to demonstrate the desired results. Similar combinations of immediate visual feedback, concreteness, and directness are present in most by-demonstration VPLs.

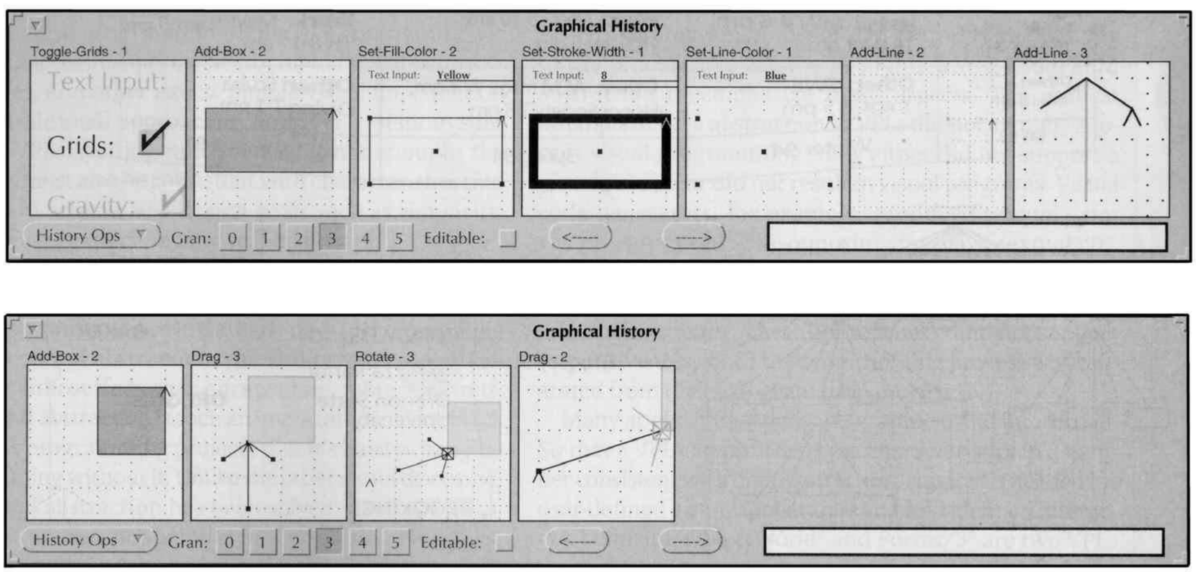

Figure 1: Programming by demonstration in Chimera. In this example, the user has drawn a box with an arrow pointing to it (as in a graph diagram), and this demonstration is depicted after-the-fact in a series of intelligently-filtered panels. This set of demonstrations can be generalized into a macro for use in creating the other nodes in the graph semi-automatically.

Form/Spreadsheet Based Visual Programming

Forms/3 [3] is an example of a VPL that follows the form-based paradigm. In this paradigm, a programmer programs by creating a form and specifying its contents. This paradigm is most commonly seen in commercial spreadsheets, in which the form is grid-shaped, and the contents are specified by the cells' formulas.

Forms/3 programs include forms (spreadsheets) with cells, but the cells are not locked into a grid.

A Forms/3 programmer creates a program by using direct manipulation to place cells on forms, and defines a formula for each cell using a flexible combination of pointing, typing, and gesturing. See Figure 2. A program's calculations are entirely determined by these formulas. The formulas combine into a network of (one-way) constraints, and the system continuously ensures that all values displayed on the screen satisfy these constraints.

Forms/3 is a Turing-complete language. The aim is to enhance the use of ordinary spreadsheet concepts to support the advanced functionality needed for full-featured programming. Thus it supports such features as graphics, animation, and recursion, but without resorting to state-modifying macros or links to traditional programming languages. For example, Forms/3 supports a rich and extensible collection of types by allowing attributes of a type to be defined by formulas, and an instance of a type to be the value of a cell, which can be referenced just like any cell. In Figure 2, an instance of type "box" is being specified by graphically sketching it; this specification can be changed if necessary by stretching the box by direct manipulation. Immediate visual feedback at liveness level 4 is provided in either case. Concreteness is present in the fact that the resulting box is immediately seen when enough formulas have been provided to make this possible; directness is present in the direct-manipulation mechanism for specifying a box because one demonstrates the specification directly on the box.

The intended audience for Forms/3 is "future" programmers--those whose job will be to create applications, but whose training has not emphasized today's traditional programming languages. A goal of Forms/3 has been to reduce the number and complexity of the mechanisms required to do application programming, with the hope that greater ease of use by programmers will result than has been characteristic of traditional languages, with an accompanying increase in correctness and/or speed of programming. In empirical studies, programmers have demonstrated greater correctness and speed in both program creation and program debugging using Forms/3's techniques than when using a variety of alternative techniques [3, 7, 18].

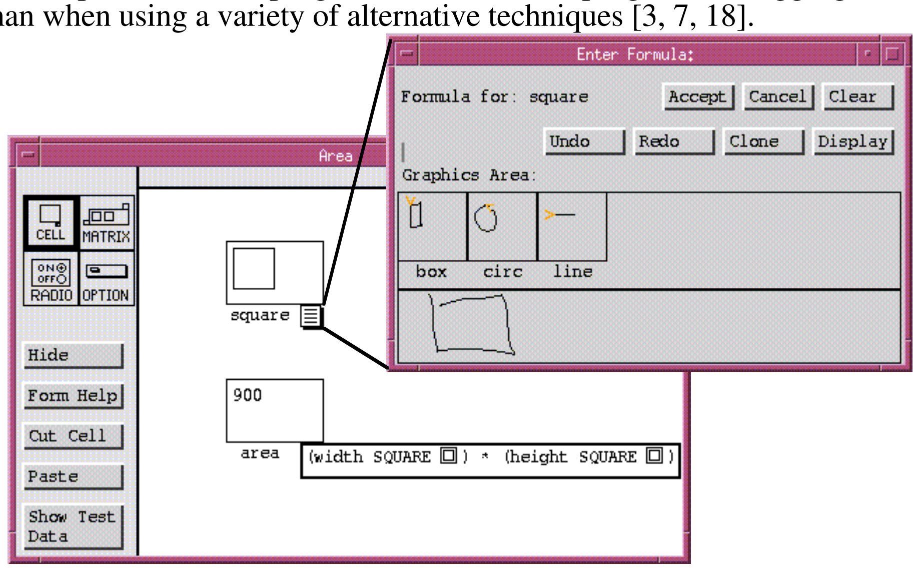

Figure 2: Defining the area of a square using spreadsheet-like cells and formulas in Forms/3. Graphical types are supported as first-class values, and the programmer can enter cell square's formula either by sketching a square box or by typing textual specifications (e.g., "box 30 30").

Dataflow Visual Programming

Prograph [9] is a dataflow VPL aimed at professional programmers. The dataflow paradigm is currently the approach to visual programming used most widely in industry. Prograph exemplifies its use for programming at all levels, from low-level details that can be grouped into procedures and objects (see Figure 3), to compositions of procedures and objects. The dataflow paradigm is also commonly used by domain-specific VPEs for composition of low-level components that have been written some other way; for example, scientific visualization systems and simulation systems often make heavy use of visual dataflow programming.

Prograph provides strong debugging support by making extensive use of dynamic visualization techniques. The liveness level is 2 for the data values themselves--the programmer explicitly requests display of a value each time he/she wants to see it. However, the runtime stack activity and the order in which nodes fire can be viewed throughout execution, and if the programmer changes a bit of data or source code mid-execution, the stack window and related views automatically adjust to proceed from that point on under the new version, and this aspect is liveness level 3.

One way in which the dataflow paradigm distinguishes itself from many other paradigms is through its explicitness (through the explicit rendering of the edges in the graph) about the dataflow relationships in the program. Since many dataflow languages govern even control flow by dataflow, these edges are also sufficient to reflect control flow explicitly in a purely dataflow language.

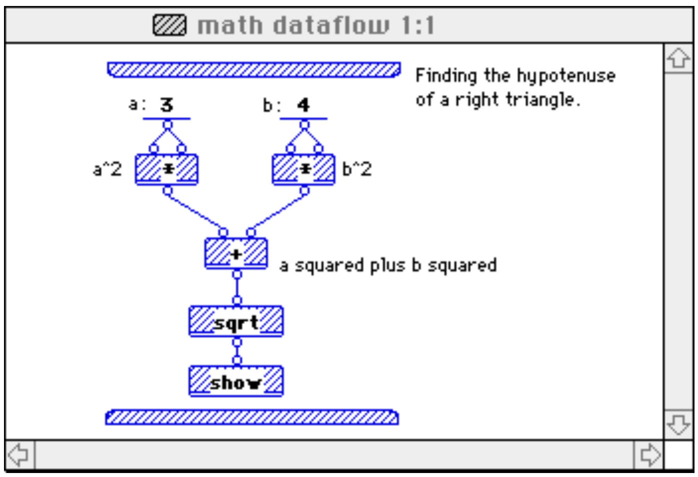

Figure 3: Dataflow programming in Prograph. Here the programmer is using the low-level (primitive) operations to find the hypotenuse of a right triangle. Prograph allows the programmer to name and compose such low-level graphs into higher-level graphs that can then be composed into even higher-level graphs, and so on.

Rule-Based Visual Programming

Cocoa [20] (formerly known as KidSim) is a rule-based VPL in which the programmer specifies the rules by demonstrating a postcondition on a precondition. See Figure 4. The intended "programmers" are children, and the problem domain is specification of graphical simulations and games. Cocoa is a Turing-complete language, but its features have not been designed to make general-purpose programming convenient; rather, it has been designed to make accessible to children the ability to program their own simulations.

The way concreteness and directness are seen in Cocoa is quite similar to Chimera, since both use by-demonstration as the way semantics are specified. The liveness level is different though; in Cocoa, liveness is between level 2 and level 3. It is not level 3 for some kinds of program changes (e.g., addition of new rules) that do not affect the current display of variables until the child requests that the program resume running, but for other kinds of program changes (e.g., changing the appearance of an object), the changes are automatically propagated into the display immediately.

In listing the properties common to rule-based systems, Hayes-Roth includes the ability to explain their behavior [12]. In Cocoa, a child can open (by selecting and double-clicking) any character participating in the simulation, and a window containing the rules governing that character's behavior is displayed, as in the figure. In each execution cycle, each character's rules are considered top-down in the character's list. The indicators next to each rule are "off" (gray) prior to a rule being considered. Then, if the rule-matching fails, the indicator next to the rule turns red; if the pattern-matching succeeds, the rule fires, the indicator next to it turns green. Once a rule has fired for a character, that character's "turn" if over, and no more rules for that character are checked until the next cycle.

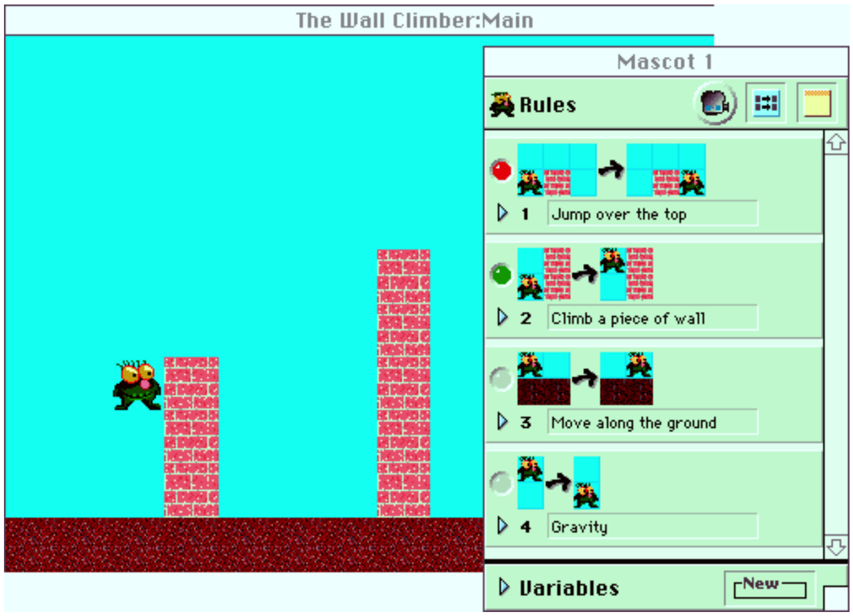

*Figure 4: A Cocoa wall-climber (*The Wall Climber: Main *window) is following the rules (*Mascot 1 *window) that have been demonstrated for it. Each rule is shown with the graphical precondition on the left of the arrow and the graphical postcondition on the right of the arrow. The wall climber has just finished following rule 2, which places it in a position suitable for following rule 1 next.*

Advanced Issues

Visual Programming and Abstraction

One of the challenges in visual programming research is scaling up to the support of ever-larger programs. This is a greater issue for VPLs than for traditional textual languages (although it certainly can be said to exist in both) for reasons relating to representation, language design and implementation, and relative youth of the area. For example, some of the visual mechanisms used to achieve characteristics such as explicitness can occupy a great deal of space, making it harder to maintain context. Also, it is hard to apply in a straightforward way techniques developed for traditional languages, because doing so often results in a reintroduction of the very complexities VPLs have tried to remove or simplify.

Recent developments in the area of abstraction have been particularly important to the scalability of VPLs. The two most widely-supported types of abstraction, both in visual and textual languages, are procedural abstraction and data abstraction. In particular, procedural abstraction has shown itself to be supportable by a variety of VPLs. A key attribute to supporting procedural abstraction in a VPL has been consistency with the rest of programming in the same VPL. Representative solutions include allowing the programmer to select, name, and iconify a section of a dataflow graph (recall Figure 3), which adds a node representing the subgraph to a library of function nodes in a dataflow language; setting up separate spreadsheets (recall Figure 2), which can be automatically generalized to allow user-defined “functions” in a form-based language; and recording and generalizing a sequence of direct manipulations (recall Figure 1) in a by-demonstration language.

Data abstraction has been slower in coming to VPLs, largely because it is sometimes difficult to find a way to maintain characteristics such as concreteness or feedback, while adding support for ideas central to data abstraction such as generality and information hiding. Still, support for data abstraction has emerged for a number of VPLs. For example, in Forms/3, a new data type is defined via a spreadsheet, with ordinary cells defining operations or methods, and with two distinguished cells that allow composition of complex objects from simpler ones and definition of how an object should appear on the screen. In Cocoa, each character’s appearance is painted using a graphical editor, and each demonstration of a new rule “belongs” to the character type being manipulated, providing roughly the functionality of an operation or method. Both Forms/3 and Cocoa also support limited forms of inheritance.

Visual Programming Language Specification

The one-dimensionality of traditional textual languages means that there is only one relationship possible between symbols of a sentence, “next to”. Thus, in describing a textual language in BNF, it is necessary to specify only the symbols in the language, not the relationship “next to” (which is implied when one symbol is written next to another in a grammar). However, the multidimensionality of VPLs means many relationships are possible, such as “overlaps,” “touches,” and “to the left of,” and there is no universally-agreed-upon definition of exactly when such relationships hold, or even how many of them may hold simultaneously between the same symbols. Hence, relationships among symbols cannot be left implicit, and traditional mechanisms such as BNF for specifying textual languages cannot be used without modification for specifying VPLs.

Many different formalisms for the specification of visual languages have been investigated. Grammar-like formalisms range from early approaches like web and array grammars and shape grammars to recent formalisms like positional grammars, relation grammars, unification grammars, attributed multiset grammars, and several types of graph grammars. There are also some non grammar-like formalisms. One grammar approach is constraint multiset grammars (CMGs) [15]. An example of a CMG production taken from the specification of state diagrams is:

```
TR:transition ::= A:arrow, T:text
        where exists R:state, S:state where
        T.midpoint close_to A.midpoint,
        R.radius = distance(A.startpoint, R.midpoint),
        S.radius = distance(A.endpoint, S.midpoint)
        and TR.from=R.name, TR.to=S.name, TR.label=T.string.
```

In general, in CMGs, each production has the form:

$$x ::= X_1, ..., X_n \textit{ where exists } X_1',...,X_m' \textit{ where } C \textit{ then } \vec{v}=E$$

meaning that the non-terminal x can be rewritten to the multiset $X_1, ..., X_n$ if the sentence contains symbols $X_1',...,X_m'$ (the context) such that the attributes of these symbols satisfy the constraint C. $\vec{v}$ denotes the vector of attributes of x whose values are defined by the vector expression E over attributes of other objects in the production. In the above example, $\vec{v}$ = (TR.from, TR.to, TR.label) and E = (R.name, S.name, T.string).

Marriott and Meyer have used the CMG approach to derive a Chomsky-like taxonomy for VPLs [15]. To show that the generality of the taxonomy is not dependent on its roots in CMGs, they also showed how several of the other formalisms can be mapped to CMGs.

Visual Programming and Cognitive Theory

Since the goals of VPLs have to do with improving humans' ability to program, it is important to consider what is known about cognitive issues relevant to programming. Much of this information has been gleaned in the field of cognitive psychology, and psychologist Thomas Green and his colleagues have made many of these findings available to non-psychologists through *cognitive dimensions* [11], a set of terms describing the structure of a programming language's components as they relate to cognitive issues in programming.

Table 1 lists the dimensions, along with a thumb-nail description of each. The relation of each dimension to a number of empirical studies and psychological principles is given in [11], but the authors also carefully point out the gaps in this body of underlying evidence. In their words, "The framework of cognitive dimensions consists of a small number of terms which have been chosen to be easy for non-specialists to comprehend, while yet capturing a significant amount of the psychology and HCI of programming."

Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on "How am I doing"?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Secondary notation	Can programmers use layout, color, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Table 1: The cognitive dimensions.

A concrete application of the cognitive dimensions is *representation design benchmarks* [24], a set of quantifiable measurements that can be made on a VPL's static representation. The benchmarks are of three sorts: (1) binary (yes/no) measurements reflecting the presence (denoted S_p) of the elements of a static representation S, (2) measurements of the extent of characteristics (denoted S_c) in a VPL's static representation, or (3) number of user navigational actions (denoted NI) required to navigate to an element of the static representation if it is not already on the screen. The benchmarks are given in Table 2.

Benchmark Name	S_c	S_p	NI	Aspect of the Representation	Computation
D1		x		Visibility of dependencies	(Sources of dependencies explicitly depicted) / (Sources of dependencies in system)
D2			x		The worst case number of steps required to navigate to the display of dependency information
PS1		x		Visibility of program structure	Does the representation explicitly show how the parts of the program logically fit together? Yes/No
PS2			x		The worst case number of steps required to navigate to the display of the program structure
L1		x		Visibility of program logic	Does the representation explicitly show how an element is computed? Yes/No
L2			x		The worst case number of steps required to make all the program logic visible
L3	x				The number of sources of misrepresentations of generality
R1		x		Display of results with program logic	Is it possible to see results displayed statically with the program source code? Yes/No
R2			x		The worst case number of steps required to display the results with the source code.
SN1		x		Secondary notation: non-semantic devices	SNdevices / 4 where SNdevices = the number of the following secondary notational devices that are available: optional naming, layout devices with no semantic impact, textual annotations and comments, and static graphical annotations.
SN2			x		The worst case number of steps to access secondary notations
AG1		x		Abstraction gradient	AGsources / 4 where AGsources = the number of the following sources of details that can be abstracted away: data details, operation details, details of other fine-grained portions of the programs, and details of NI devices.
AG2			x		The worst case number of steps to abstract away the details
RI1		x		Accessibility of related information	Is it possible to display all related information side by side? Yes/No
RI2			x		The worst case number of steps required to navigate to the display of related information.
SRE1	x			Use of screen real estate	The maximum number of program elements that can be displayed on a physical screen.
SRE2	x				The number of non-semantic intersections on the physical screen present when obtaining the SRE1 score
AS1, AS2, AS3	x x x			Closeness to a specific audience's background	ASyes's / ASquestions where ASyes's = the number of "yes" answers, and ASquestions = the number of itemized questions of the general form: "Does the <representation element> look like the <object/operation/ composition mechanism> in the intended audience's prerequisite background?"

Table 2: Summary of the representation design benchmarks. S_c denotes measures of the characteristics of elements of S. S_p denotes measures of the presence of potential elements of S. Each S_p measure has a corresponding NI measure.

Empirical Findings

Work toward using visual programming techniques to improve correctness and/or speed in programming tasks has focused primarily on three areas: program comprehension, program creation, and program debugging. Of these three areas, the most empirical studies have been done on VPLs' effects on program comprehension. See [22] for a survey of this work. The results of these studies have been mixed, reporting findings for some kinds of programs or audiences in which VPLs and/or visual notations are linked with greater comprehension, and others in which strictly textual languages and/or notations have been linked with greater comprehension.

There have been fewer empirical studies on program creation thusfar, but these studies have produced far more consistent results than the studies on comprehension. Most have reported visual approaches outperforming traditional textual approaches for this task [1, 3, 16, 18].

Finally, the effects of visual programming are the least studied of all in debugging (and in fact this is also true of classical debuggers, which feature the precursors of the ideas of liveness as now found in VPLs). These studies have not found statistically significant improvements for all the aspects studied, but for the aspects in which statistical significance was found, visual approaches including immediate feedback were found to be superior to the static, non-feedback-oriented approaches in most cases [7, 11].

Summary

Visual programming is found in both VPLs and VPEs. Commercially, visual programming is most commonly found in VPEs, which serve as an effective conduit for some of the gains made from research in VPLs to be quickly transferred into industrial practice. The goal of visual programming in general is to programming easier for humans, and the goal of VPLs in particular is better programming language design. Such a goal is timely because today's supporting hardware and software places fewer restrictions on what elements may be part of the vocabulary of a programming language. Opportunities that arise from this reduction of restrictions that have received the most attention so far in VPL design are concreteness, directness, explicitness, and immediate visual feedback. However, exploiting these areas can mean radical departures from tradition, and this in turn requires reinvention of building blocks such as abstraction mechanisms, which are important in designing scalable VPLs. The multidimensionality inherent in VPLs also leads to language theoretic issues. Finally, the fact that VPLs are intended to make programming easier for humans leads to a need for more research about how human cognitive abilities are best served by innovations in programming language design.

Acknowledgments and Bibliographic Notes

The sources of information used for this article, other than those specifically referenced above, as well as additional sources of information, are as follows. The material for the introductory section is derived from [4]. See [5] for a detailed treatment of the scaling-up problem for visual programming languages. The four VPL examples were drawn from an IEEE tutorial presented jointly by Burnett and Rebecca Walpole Djang in 1997 in Capri, Italy. The discussion of VPL specification presented here is summarized from [15]; other approaches to formal syntax issues and also to formal semantics issues can be found in [2, 6, 8, 10, 23]. The discussions of cognitive dimensions and of representation design benchmarks are due to [24]. The summary of empirical studies is derived from [3] and from [22].

References

1. E. Baroth and C. Hartsough, Visual programming in the real world. In M. Burnett, A. Goldberg, T. Lewis (eds.), *Visual Object-Oriented Programming: Concepts and*

Environments, Prentice-Hall, Englewood Cliffs, NJ; Manning Publications, Greenwich, Connecticut; and IEEE, Los Alamitos, California, 1995.

2. P. Bottoni, M. Costabile, S. Levialdi, and P. Mussio, Visual conditional attributed rewriting systems in visual language specification. *IEEE Symposium on Visual Languages*, Boulder, Colorado: 156-163, September 3-6, 1996.
3. M. Burnett and H. Gottfried, Graphical definitions: expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction* 5(1), March 1998.
4. M. Burnett and D. McIntyre, Visual programming. *Computer* 28(3): 14-16, March 1995.
5. M. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee, Scaling up visual programming languages. *Computer* 28(3): 45-54, March 1995.
6. S. Chang, G. Tortora, B. Yu, A. Guercio, Icon purity - towards a formal definition of icons. *International Journal of Pattern Recognition and Artificial Intelligence* 1: 377-392, 1987.
7. C. Cook, M. Burnett, and D. Boom, A bug's eye view of immediate visual feedback in direct-manipulation programming systems. *Empirical Studies of Programmers: Seventh Workshop*, Alexandria, Virginia: 20-41, Oct. 24-26, 1997.
8. G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M. Tucci, Automatic parser generation for pictorial languages. *IEEE Symposium on Visual Languages*, Bergen, Norway: 306-313, August 24-27, 1993.
9. Cox, P., F. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning. *1989 IEEE Workshop on Visual Languages*, Rome, Italy, Oct. 4-6, 1989.
10. M. Erwig, Semantics of visual languages. *IEEE Symposium on Visual Languages*, Capri, Italy: 300-307, September 23-26, 1997.
11. T. Green and M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* 7(2): 131-174, June 1996.
12. F. Hayes-Roth, Rule-based systems. *Communications of the ACM* 28(9): 921-932, September 1985.
13. E. Hutchins, J. Hollan, and D. Norman, Direct manipulation interfaces. In D. Norman and S. Draper (eds.),*User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Assoc., Hillsdale, NJ: 87-124, 1986.
14. D. Kurlander, Chimera: example-based graphical editing. In A. Cypher (ed.),*Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, Mass, 1993.
15. K. Marriott and B. Meyer, On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing* 8(4): 375-402, August 1997.
16. F. Modugno, A. Corbett, and B. Myers, Evaluating program representation in a demonstrational visual shell. *Empirical Studies of Programmers: Sixth Workshop*, Alexandria, Virginia: 131-146, January 1996.
17. B. Nardi, *A Small Matter of Programming: Perspectives on End User Computing,* MIT Press, Cambridge, Mass., 1993.
18. R. Pandey and M. Burnett, Is it easier to write matrix manipulation programs visually or textually? An empirical study. *IEEE Symposium on Visual Languages,* Bergen, Norway: 344-351, August 24-27, 1993.
19. B. Shneiderman, Direct manipulation: a step beyond programming languages. *Computer* 16(8): 57-69, August 1983.
20. D. Smith, A. Cypher, and J. Spohrer, Kidsim: programming agents without a programming language. *Communications of the ACM* 37(7): 54-67, July 1994.
21. Tanimoto, S., VIVA: a visual language for image processing. *Journal of Visual Languages Computing* 2(2): 127-139, June 1990.
22. K. Whitley, Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing* 8(1), February 1997, 109-142.
23. K. Wittenburg and L. Weitzmann, Visual grammars and incremental parsing for interface languages. *IEEE Workstop on Visual Languages*, Skokie, Illinois: 111-118, October 4-6,

1990.
24. S. Yang, M. Burnett, E. DeKoven, and M. Zloof, Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing* 8(5/6): 563-599, October/December 1997.