Visual Programming Languages: A Survey

CS 263 Final Project

Marat Boshernitsan
Michael Downes

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

May 15, 1997

Abstract

Over the past ten years, frequent increases in computer processing speed and graphics display capabilities have made possible a great deal of research and experimentation in the field of visual programming languages. A variety of different design methodologies have arisen from these research efforts, and numerous visual programming systems have been developed to address both specific application areas, such as user interface design and physical simulation, and more general programming tasks. We present a survey of the field of visual programming languages beginning with an historical overview of some of the pioneering efforts in the field. In addition, we present different classifications of visual programming languages, touch on implementation and design issues, and discuss efforts to formalize the theoretical basis for visual languages. We also discuss and examine a variety of the more important projects in the field over the last few years.

[0]The authors can be reached at: {mdownes,maratb}@CS.Berkeley.EDU. An on-line version of this paper can be found on the WWW page: http://www.cs.berkeley.edu/~maratb/cs263/paper.html.

1 Introduction

From cave paintings to hieroglyphics to paintings of Campbell's soup cans, humans have long communicated with each other using images. The field of visual programming languages asks: why, then, do we persist in trying to communicate with our computers using textual programming languages? Would we not be more productive and would the power of modern computers not be accessible to a wider range of people if we were able to instruct a computer by simply drawing for it the images we see in our mind's eye when we consider the solutions to particular problems? Obviously, proponents of visual programming languages (VPLs) argue that the answer to both these questions is yes.

The questions above highlight the primary motivations for most research into VPLs. First, many people think and remember things in terms of pictures. They relate to the world in an inherently graphical way and use imagery as a primary component of creative thought [Smith 1975]. In addition, textual programming languages have proven to be rather difficult for many creative and intelligent people to learn to use effectively. Reducing or removing entirely the necessity of translating visual ideas into somewhat artificial textual representations can help to mitigate this steep learning curve problem. Furthermore, a variety of applications, including scientific visualization and interactive simulation authoring, lend themselves particularly well to visual development methods.

The sections which follow present a survey of the field of visual programming languages which has emerged in response to the issues mentioned above. We begin with a discussion of the historical trends and early work which laid the foundations for modern research in the field (Section 2). Section 3 provides a description of a taxonomy for visual languages. We follow with a primer on the main theoretical aspects of VPLs in Section 4, and briefly discuss a variety of language issues in Section 5. We then clarify the topics discussed in previous sections by presenting examples of important and interesting VPLs (Section 6). Finally, we share some concluding remarks and thoughts on future work (Section 7).

2 History of Visual Programming Languages

The field of visual programming has grown from a marriage of work in computer graphics, programming languages, and human-computer interaction. It should come as no surprise, then, that much of the seminal work in the field is also viewed as pioneering work in one of the other disciplines. Ivan Sutherland's groundbreaking Sketchpad system stands out as the best example of this trend [Sutherland 1963]. Sketchpad, designed in 1963 on the TX-2 computer at MIT, has been called the first computer graphics application. The system allowed users to work with a lightpen to create 2D graphics by creating simple primitives, like lines and circles, and then applying operations, such as copy, and constraints on the geometry of the shapes. Its graphical interface and support for user-specifiable constraints stand out as Sketchpad's most important contributions to visual programming languages. By defining appropriate constraints, users could develop structures such as complicated mechanical linkages and then move them about in real time. We will see the idea of visually specified constraints and constraint-oriented programming resurface in a number of later VPLs. Ivan Sutherland's brother, William, also made an important early contribution to visual programming in 1965, when he used the TX-2 to develop a simple visual dataflow language. The system allowed users to create, debug, and execute dataflow diagrams in a unified visual environment [Najork 1995].

The next major milestone in the genesis of VPLs came in 1975 with the publication of David Canfield Smith's PhD dissertation entitled "Pygmalion: A Creative Programming Environment" [Smith 1975]. Smith's work marks the starting point for a number of threads of research in the field which continue to this day. For example, Pygmalion embodied an icon-based programming paradigm in which the user created, modified, and linked together small pictorial objects, called icons, with defined properties to perform computations. Much work has since gone into formalizing icon theory, as will be discussed below, and many modern VPLs employ an icon-based approach. Pygmalion also made use of the concept of programming-by-example wherein the user shows the system how to perform a task in a specific case and the system uses this information to generate a program which performs the task in general cases. In Smith's system, the user sets the environment to "remember" mode, performs the computation of interest, turns off "remember" mode, and receives as output a program, in a simple assembly-like subset of Smalltalk, which performs the computation on an arbitrary input.

3 Classification of Visual Programming Languages

As the field of VPLs has matured, more and more interest has been focused on creating a robust, standardized classification for work in the area. Such a classification system not only aids researchers in finding related work but also provides a baseline with which to compare and evaluate different systems. Some of the most important names in the field, including Chang, Shu, and Burnett, have worked on identifying the defining characteristics of the major categories of VPLs [Chang 1987, Shu 1986, Burnett & Baker 1994]. The following presents a summary of the classification scheme discussed below:

1. Purely visual languages
2. Hybrid text and visual systems
3. Programming-by-example systems
4. Constraint-oriented systems
5. Form-based systems

Note that the categories are by no means mutually exclusive. Indeed, many languages can be placed in more than one category.

The single most important category has to be purely visual languages. Such languages are characterized by their reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program which is subsequently debugged and executed in the same visual environment. The program is compiled directly from its visual representation and is never translated into an interim text-based language. Examples of such completely visual systems include VIPR, Prograph, and PICT, the first two of which will be discussed in more detail below. In much of the literature in the field, this category is further subdivided into sections like iconic and non-iconic languages, object-oriented, functional, and imperative languages [Chang 1987, Burnett & Baker 1994]. However, for our purposes a slightly larger granularity helps to emphasize the major visually-oriented differences between various VPLs.

One important subset of VPLs attempts to combine both visual and textual elements. These hybrid systems include both those in which programs are created visually and then translated into an underlying high-level textual language and systems which involve the use of graphical elements in an otherwise textual language. Examples in this category include Rehearsal World and work by Erwig et. al. In the former, the user trains the system to solve a particular problem by manipulating graphical "actors," and then the systems generates a Smalltalk program to implement the solution [Finzer & Gould 1984]. The latter involves work on developing extensions to languages like C and C++ which allow programmers to intersperse their text code with diagrams [Erwig & Meyer 1995]. For instance, one can define a linked list data structure textually and then perform an operation like deletion of a node by drawing the steps in the process.

In addition to these two major categories, many VPLs fall into a variety of smaller classifications. For example, a number of VPLs follow in the footsteps of Pygmalion by allowing the user to create and manipulate graphical objects with which to "teach" the system how to perform a particular task. Rehearsal World, described above, fits into this category of programming by example. Some VPLs can trace their lineage back, in part, to Sutherland's constraint manipulations in Sketchpad. These constraint-oriented systems are especially popular for simulation design, in which a programmer models physical objects as objects in the visual environment which are subject to constraints designed to mimic the behavior of natural laws, like gravity. Constraint-oriented systems have also found application in the development of graphical user interfaces. Thinglab and ARK, both primarily simulation VPLs, stand out as quintessential examples of constraint-based languages [Smith 1986, Borning 1981]. A few VPLs have borrowed their visualization and programming metaphors from spreadsheets. These languages can be classified as form-based VPLs. They represent programming as altering a group of interconnected cells over time and often allow the programmer to visualize the execution of a program as a sequence of different cell states which progress through time [Burnett & Ambler 1992]. Forms/3 is the current incarnation of the progenitor of this type of VPL, and it will be covered in detail below. It is important to note that in each of the categories mentioned above, we can find examples of both general-purpose VPLs and languages designed for domain-specific applications.

The field of visual programming has evolved greatly over the last ten years. Continual development and refinement of languages in the categories discussed above have led to some work which was initially considered to be part of the field being reclassified as related to but not actually exemplifying visual programming. These VPL orphans, so to speak, include algorithm animation systems, such as BALSA [Brown & Sedgewick 1984], which provide interactive graphical displays of executing programs and graphical user interface development tools, like those provided with many modern compilers including Microsoft Visual C++. Both types of systems certainly include highly visual components, but they are more graphics applications and template generators than actual programming languages.

4 Theory of Visual Programming Languages

In this section, we survey the theoretical advances in the field of Visual Programming Languages, mostly derived from early work by S.-K. Chang on generalized icon theory. To set up the framework for the discussion which follows, we put forth some definitions from [Chang 1990]:

icon (generalized icon)
: An object with the dual representation of a logical part (the meaning) and a physical part (the image).

iconic system
: A structured set of related icons.

iconic sentence (visual sentence)
: A spatial arrangement of icons from iconic system.

visual language
A set of iconic sentences constructed with given syntax and semantics.

syntactic analysis (spatial parsing)
An analysis of an iconic sentence to determine the underlying structure.

semantic analysis (spatial interpretation)
An analysis of an iconic sentence to determine the underlying meaning.

In this section, we restrict our discussion to two-dimensional visual languages, although everything that follows can be generalized to three (and more).

4.1 Formal Specification of Visual Programming Languages

A spatial arrangement of icons that constitutes a visual sentence is a two-dimensional counterpart of a one-dimensional arrangement of tokens in conventional (textual) programming languages. In those languages, a program is expressed as a string in which terminal tokens are concatenated to form a sentence whose structure and meaning are discovered by syntactic and semantic analysis, respectively. Thus, the construction rule is implicit in the language and need not be spelled-out as part of the language specification. Conversely, in visual programming languages we distinguish three construction rules that are used to spatially arrange icons: horizontal concatenation (denoted by `&`), vertical concatenation (denoted by `^`), and spatial overlay (denoted by `+`).

In formalizing visual programming languages, it is customary to distinguish *process icons* from *object icons*. The former express computations; the latter can be further subdivided into *elementary object icons* and *composite object icons*. Elementary object icons identify primitive objects in the language, whereas composite object icons identify objects formed by a spatial arrangement of elementary object icons. Finally, the term *elementary icons* is used to refer to both process icons and elementary object icons and denotes those icons that are primitives in the language. Since a picture (or, icon, in our case) is worth a thousand words, we attempt to illustrate all of the above concepts in Figure 1 which demonstrates a few icons from the Heidelber icons set [Rhor 1986] and a complete visual sentence.

A visual programming language is specified by a triple (ID, G_0, B), where ID is the icon dictionary, G_0 is a grammar, and B is a domain-specific knowledge base [Tortora 1990]. The icon dictionary is the set of generalized icons each of which is represented by a pair (X_m, X_i), with a logical art X_m (the meaning) and a physical part X_i (the image). The grammar G_0 specifies how composite object icons may be constructed from elementary icons by using spatial arrangement operators. Note that we need to specify spatial composition operators as terminals in the grammar precisely because they are no longer implicit in the language definition. The knowledge base B contains domain-specific information necessary for constructing the meaning of a given visual sentence. It contains information regarding event names, conceptual relations, names of resulting objects, and references to the resulting objects.

4.2 Analysis of Visual Programming Languages

As discussed above, visual sentences are constructed from elementary icons using iconic operators. The syntactic analysis of visual sentences (also known as spatial parsing [Lakin 1986]) is based upon a number of approaches [Chang 1990]. Here, we present a partial listing of such approaches.

Picture-processing grammars
Originally designed to parse digital pictures on a square grid, these grammars are based on

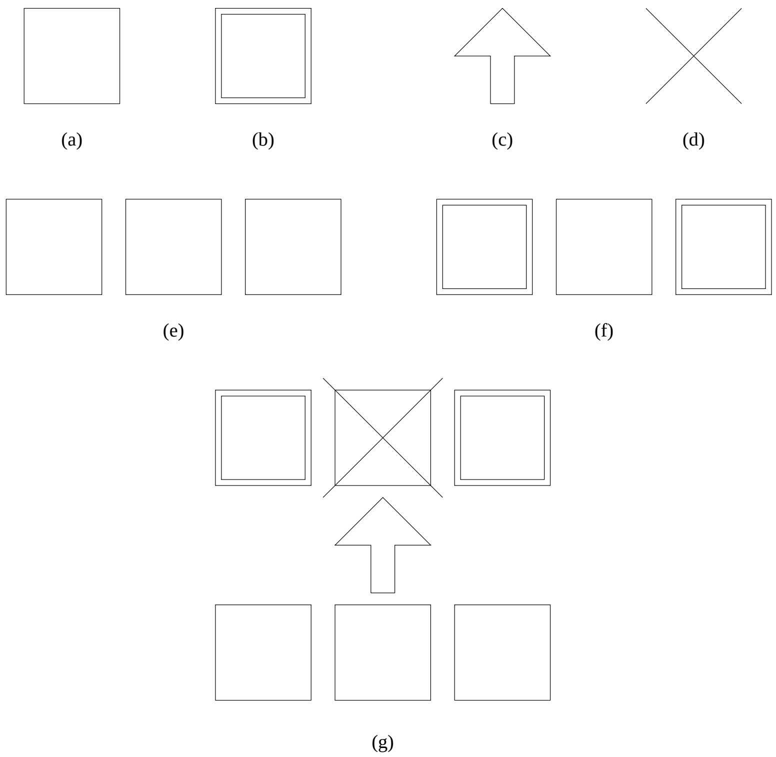


Figure 1: Some icons from the Heidelberg icon set. Elementary object icons: (a) a character, and (b) a selected character. Process icons: (c) insertion operation, and (d) deletion operation. Composite object icons: (e) a string (composed of characters), and (f) a selected string (composed of a character and two selected characters). Visual sentence denoting replacement of a substring in a string.

the fact that digital pictures are composed of pixels. These grammars discover the structure of visual sentence by composing individual pixels into recognizable visual elements (lines, arks, etc.) [Golin 1990]. This approach is useful when an iconic system needs to be able to recognize icons with a certain level of error tolerance (e.g. handwritten digits).

Precedence grammars
This spatial parsing grammar can be used for two-dimensional mathematical expression analysis and printed-page analysis. Precedence grammars are more suitable for syntactic analysis of visual sentences constructed from elementary icons and iconic operators. The parse tree is constructed by comparing precedences of operators in a pattern and subdiving the pattern into one or more subpatterns.

Context-free and context-dependent grammars
These grammars are used to specify composition of visual sentences using familiar formalisms, and so many standard methods of parsing such grammars are applicable.

Graph grammars
These are by far the most powerful (albeit least efficient) specifications of visual languages. These formalisms provide for the most means for establishing context relationships and much recent work has been devoted to making parsing with graph grammars computationally feasible [Rekers & Schürr 1995].

A parse tree produced by one of the above parsing methods is subsequently analyzed using traditional approaches to semantic analysis (e.g. attribute grammars, ad-hoc tree computations, etc.).

Because the field of visual programming languages has only recently entered a more or less mature stage, much of the work on formalization of visual languages is still in its infancy (published within last 2-3 years) and so we do not survey it here in any detail.

5 Visual Language Issues

We now discuss some common language issues in light of which the following presentation of visual languages is cast [Burnett 1994]. These issues are mostly applicable to *general-purpose visual languages* (suitable for producing executable programs of reasonable size), although certain issues will also be relevant to *domain-specific languages* (designed to accommodate a particular domain such as software engineering or scientific visualization).

5.1 Control Flow

Similarly to conventional programming languages, visual languages embrace two notions of flow of control in programs: imperative and declarative.

With the imperative approach, a visual program constitutes one or more control-flow or dataflow diagrams which indicate how the thread of control flows through the program. A particular advantage of such approach is that it provides an effective visual representation of parallelism. A disadvantage of this method is that a programmer is required to keep track of how sequencing of operations modifies the state of the program, which is not always an intended feature of the system (especially if it is designed to accommodate novices).

An alternative to imperative semantics of flow control is to use a declarative style of programming. With this approach, one only needs to worry what computations are performed, and not how the actual operations are carried out. Explicit state modification is avoided by using single assignment: a programmer creates a new object by copying an existing one and specifying the desired differences, rather than modifying the existent object's state. Also, instead of specifying a sequence of state changes, the programmer defines operations by specifying object dependencies. For example, if the programmer defines Y to be $X + 1$, this explicitly states that Y is to be computed using object in X, allowing the system to infer that X's value needs to be computed first. Thus, the sequencing of operations is still present, but must be inferred by the system rather than defined by the programmer. Off course, special care must be taken by the system that circular dependencies are detected and signaled as errors.

5.2 Procedural Abstraction

We distinguish two levels of procedural abstraction. High-level visual programming languages are not complete programming languages, i.e. it is not possible to write and maintain an entire program in such as language and inevitably there's some underlying non-visual modules that are combined using a visual language. This approach to visual programming is found in various domain-specific systems such as software maintenance tools and scientific visualization environments. At the opposite end of the scale, are low-level visual languages which do not allow the programmer to combine fine-grained logic into procedural modules. This methodology is also useful in various domain-specific languages such as logic simulators. General-purpose visual programming languages normally cover the entire spectrum of programming facilities ranging from low-level features, including conditionals, recursion, and iteration, to high-level facilities that allow one to combine low-level logic into abstract modules (procedures, classes, libraries, etc.).

5.3 Data Abstraction

Data abstraction facilities are only found in general-purpose programming languages. The notion of data abstraction in visual programming is very similar to the notion of data abstraction in conventional programming languages, with the only requirements being that abstract data types be defined visually (as opposed to textually), have a visual (iconic) representation, and provide for interactive behavior.

6 Visual Programming Languages

In this section we present a sample of visual programming languages that illustrate many of the concepts presented in this paper.

6.1 ARK

More than 10 years after its inception, the Alternate Reality Kit (ARK), designed by R. Smith at Xerox PARC, remains one of the more unique and visionary domain-specific VPLs. ARK, implemented in Smalltalk-80, provides users with a 2D animated environment for creating interactive simulations. The system is intended to be used by non-expert programmers to create simulations and by an even wider audience to interact with the simulations. In order to help users to understand the fundamental laws of nature, ARK uses a highly literal metaphor in which the user controls an on-screen hand which can interact with physical objects, like balls and blocks, which possess masses and velocities and with objects, called interactors, representing physical laws, like gravity [Smith 1986]. By giving a kind of physical reality to abstract laws, the system attempts

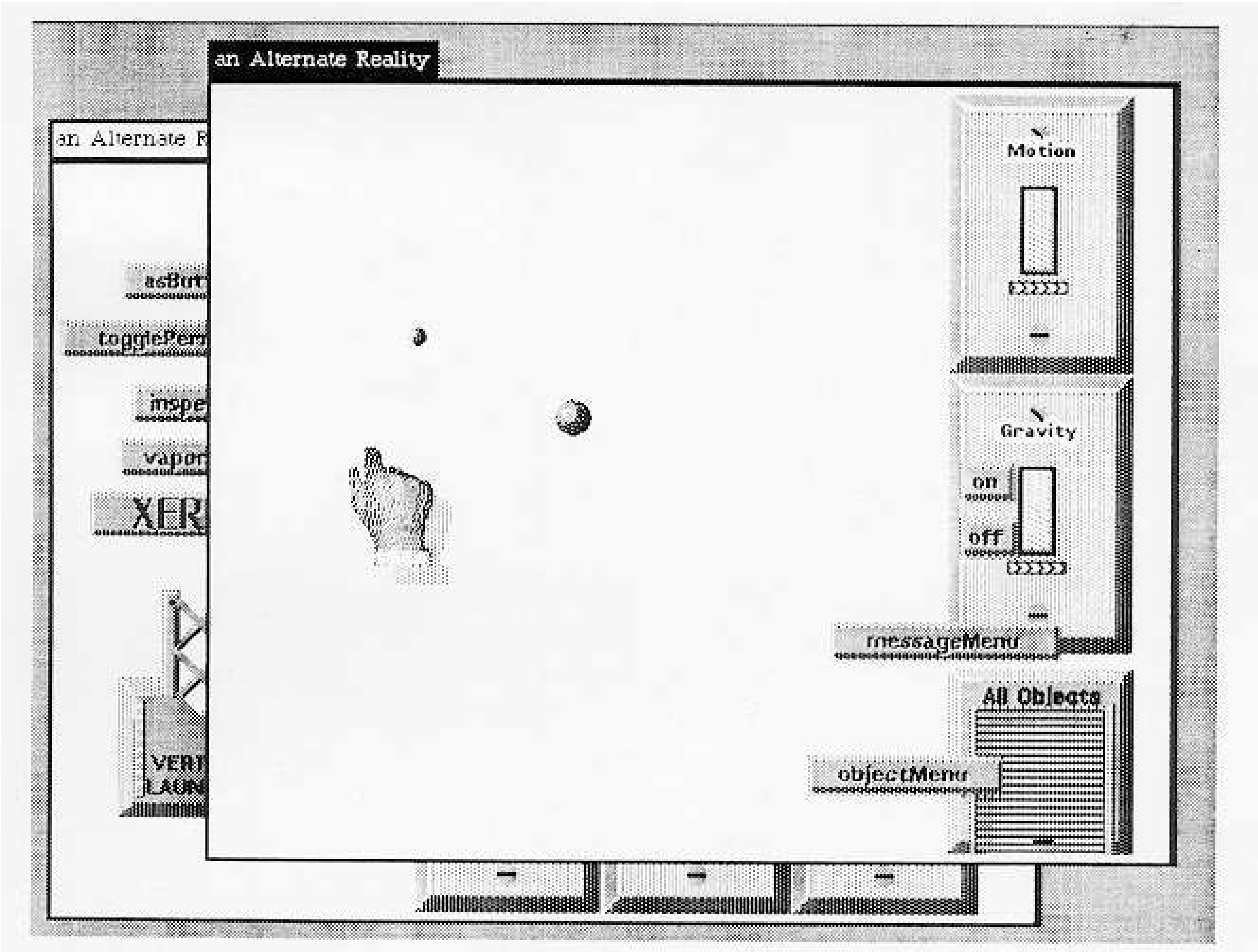


Figure 2: Planetary Orbit Simulation in ARK

to remove some of the mystery surrounding the ways in which such laws interact with objects and each other. Users can modify any objects in the environment using constructs called message boxes and buttons, viewing the results of their changes in real time. The simulation runs in an "alternate reality" contained within a window inside an all-encompassing "meta-reality." The structure is very much like a modern windows-and-desktop GUI. The programmer can move the hand between alternate realities and pull objects out of the simulation and into meta-reality at anytime. An object which has been lifted out of its alternate reality does not participate in the simulation until it is dropped out of meta-reality.

The example of a user reaching into and removing an object from an alternate reality highlights one of the more interesting design issues in ARK, namely the necessity to occasionally break with the highly literal physical world metaphor in order to provide useful functionality. Smith refers to this issue as the tension between magic and literalism in ARK [Smith 1987]. While using a hand which can grab physical objects is a highly literal component of the system, allowing a user to reach into a simulation and alter or remove objects with no regard to the physical laws currently at work in the environment clearly provides the user with what could be considered "magical" powers. The question of when to allow a magical event or action in ARK to conflict with the physical metaphor parallels a similar concern in the design of more traditional VPLs. In developing most VPLs, researchers have had to decide on the appropriate uses of text in their system. While it is possible to design a system which uses no text whatsoever, and such systems have been created, the resulting programs are often very difficult to read and understand. Most VPLs, even those which are completely visual, use text, at the very least, to label variables and functions in programs. Thus designers must face the same problem as was addressed in ARK. They must attempt to balance consistency of visual representation with usability.

Although ARK targets a fairly specific application domain, it supports a powerful programming model. Programmers can not only create simulations by linking together various pre-built objects and interactors, but they can also develop new interactors. Programming a simulation, such as the planetary orbit simulation shown in Figure 2, involves first generating physical objects, like balls, from the object warehouse in the lower right corner of the display. By clicking on the objectMenu button, the programmer can choose to instantiate any object available in the environment. After creating some physical objects, the programmer follows the same procedure to place interactors, like the Motion and Gravity objects in Figure 2, in the alternate reality. The programmer now uses the messageMenu button to find out what sorts of messages to which the various interactors respond by placing the button on an interactor and pressing it with the hand. This generates a list of all messages appropriate to that interactor. The programmer chooses one, such as "off" for the gravity interactor in the orbit simulation, and the system generates a message box. Message boxes are objects which can send and receive Smalltalk messages. The programmer links the new message box, in our case one which generates the message "off", to the appropriate interactor by joining them with a dotted line. The message box can then be collapsed to a single button as in Figure 2. Interactors affect all the objects in the same alternate reality, so after specifying all the necessary controls, a programmer can begin the simulation.

It is important to note that all of the objects in the underlying Smalltalk environment are available to the ARK programmer. Objects which are not ARK-specific appear as representative objects, like the TwoVector object in Figure 3. As shown in the figure, such Smalltalk objects can be linked with ARK objects in the same way as native objects. The example shown involves using a TwoVector object as the input to a button which sets the velocity of a disk.

Clearly, ARK interactors behave much like constraints on the physical objects in the alternate reality. Thus, creating and modifying interactors exemplify ARK's constraint-oriented features. A programmer can generate new interactors by creating networks of message boxes. As a simple

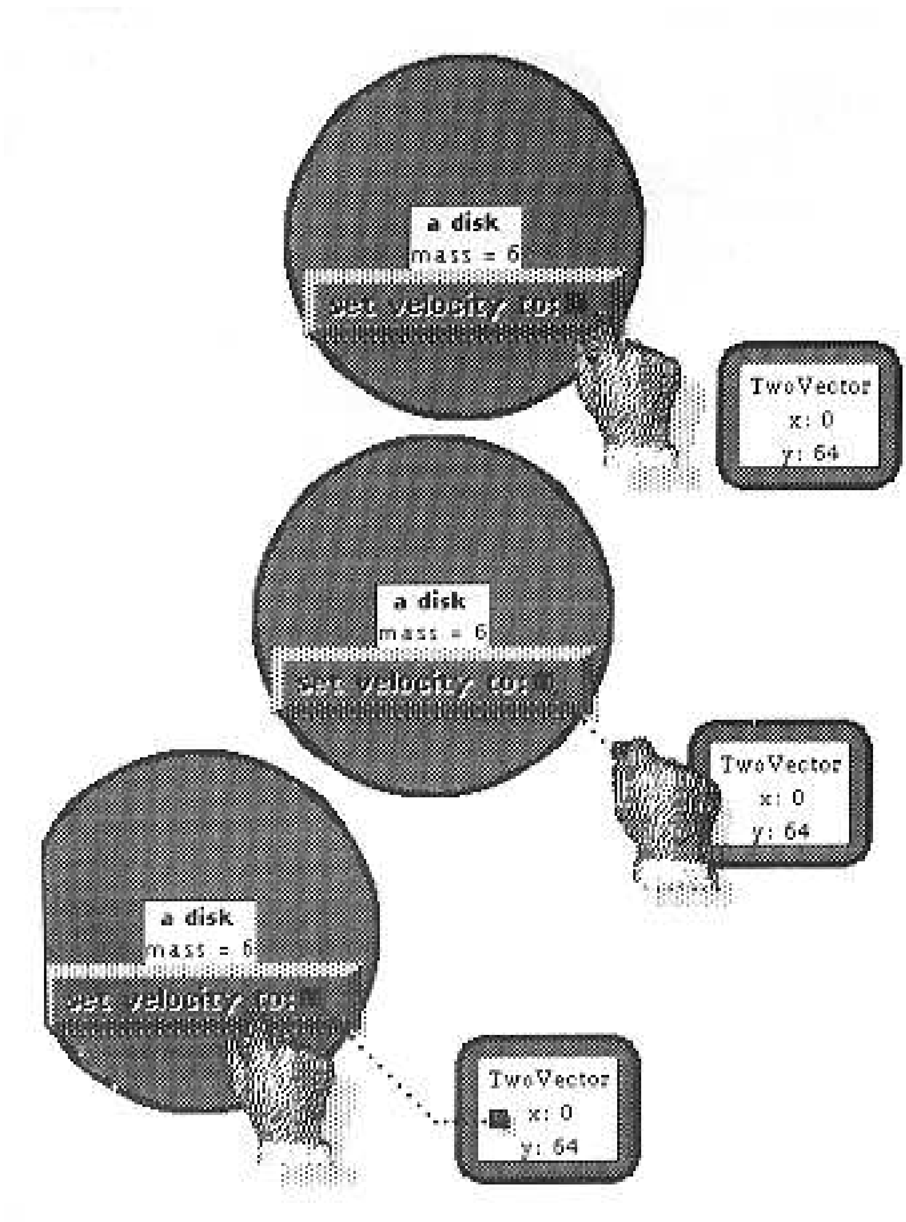


Figure 3: Accessing a Smalltalk object in ARK

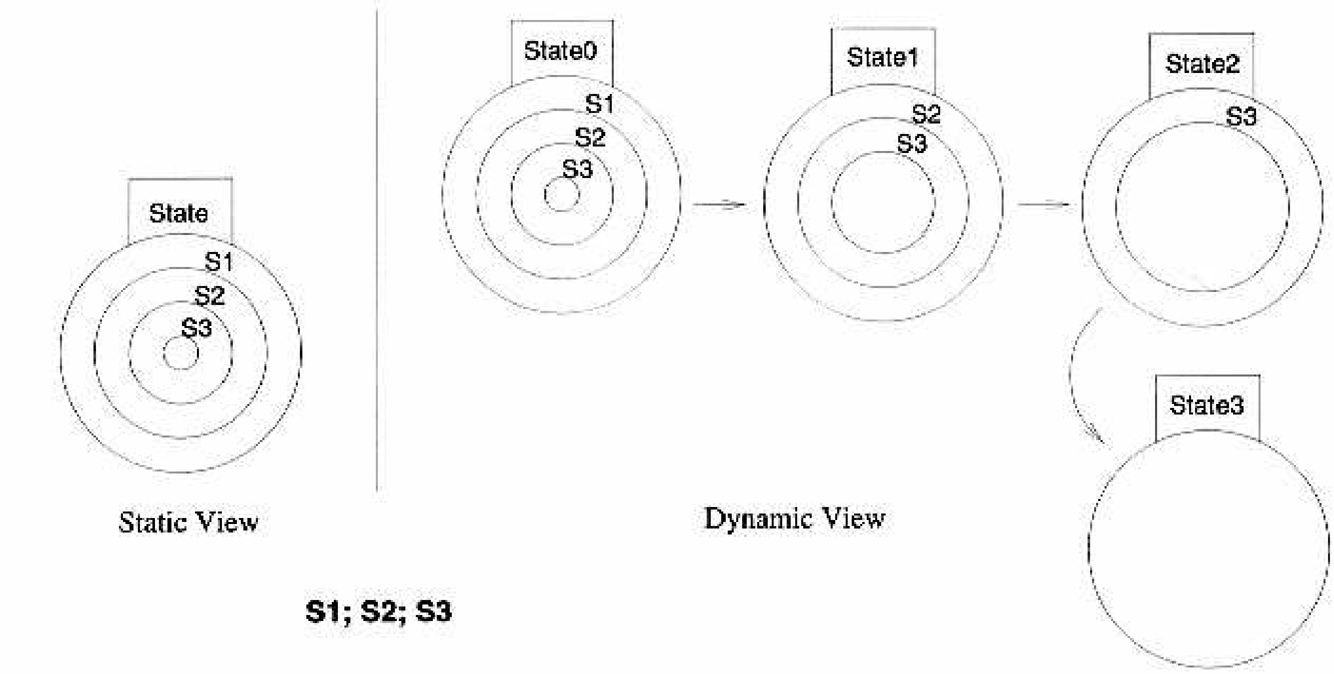


Figure 4: Visualization of program execution in VIPR

example, consider developing a frictional force interactor by creating a message box which adds a force to an object proportional to the negative of its velocity [Smith 1987]. The message box can be set to continuously send its message, and when its behavior has been verified, the programmer can convert it to an interactor.

6.2 VIPR

VIPR, or Visual Imperative PRogramming, developed by Citrin et. al at the University of Colorado represents a unique approach to completely visual general purpose programming. Rather than relying on icons, forms, or other traditional graphical representations, VIPR uses nested series of concentric rings to visualize programs, as shown in Figure 4. Each step in a computation involves merging two rings in the presence of a state object which is connected to the outermost ring. One can visualize program execution as walking down a network of pipes which branches off in different directions while changing the state based on actions written on the inside of the pipes [Citrin et al. 1994].

The ongoing development of VIPR has been motivated, in part by a desire to create an object-oriented language which is relatively easy to learn and use. As a result, VIPR includes most of the common attributes of object-oriented languages, including inheritance, polymorphism, and dynamic dispatch. The language's semantics have been defined to be similar to C++ in order to make it easier for experienced programmers to read and understand VIPR programs. However, the language is entirely visual, so the semantics of a VIPR program can be understood by applying a small number of graphical rewrite rules. This relationship to C++ also means that VIPR provides

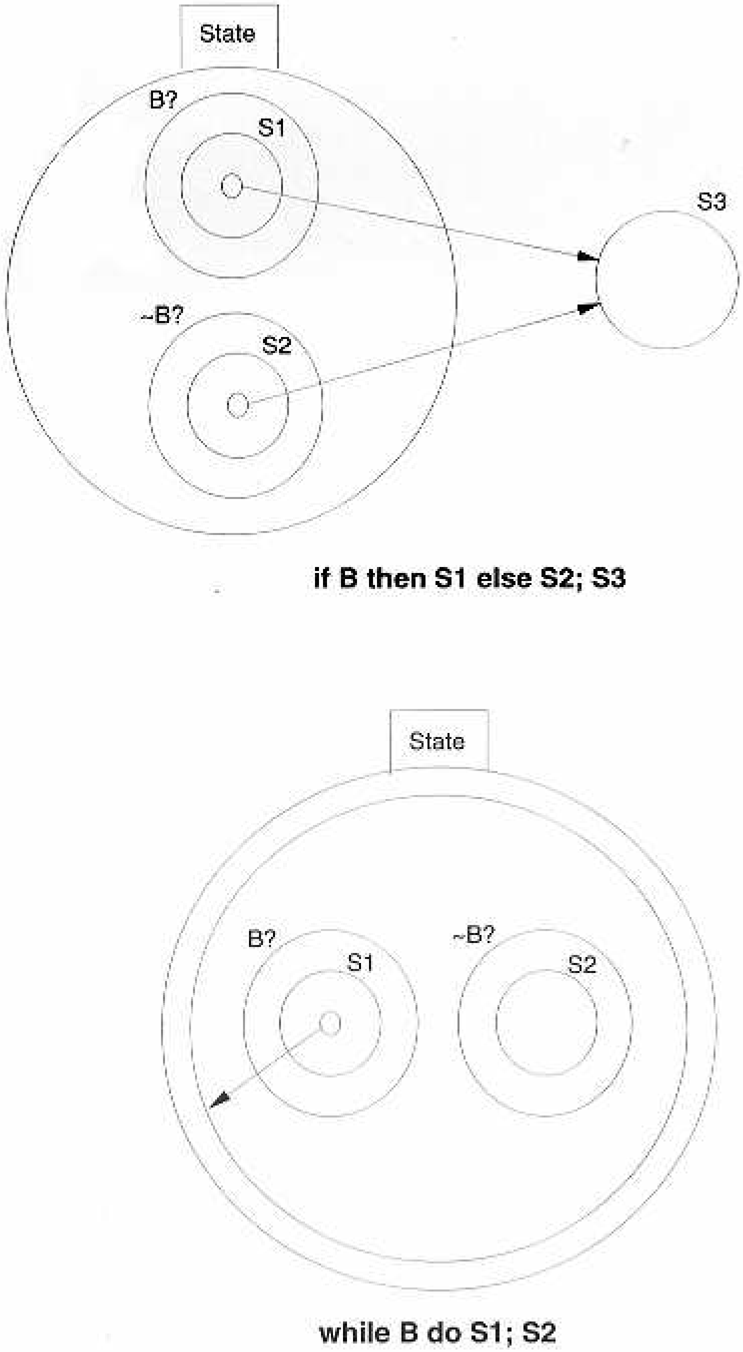


Figure 5: Example Control Constructs in VIPR

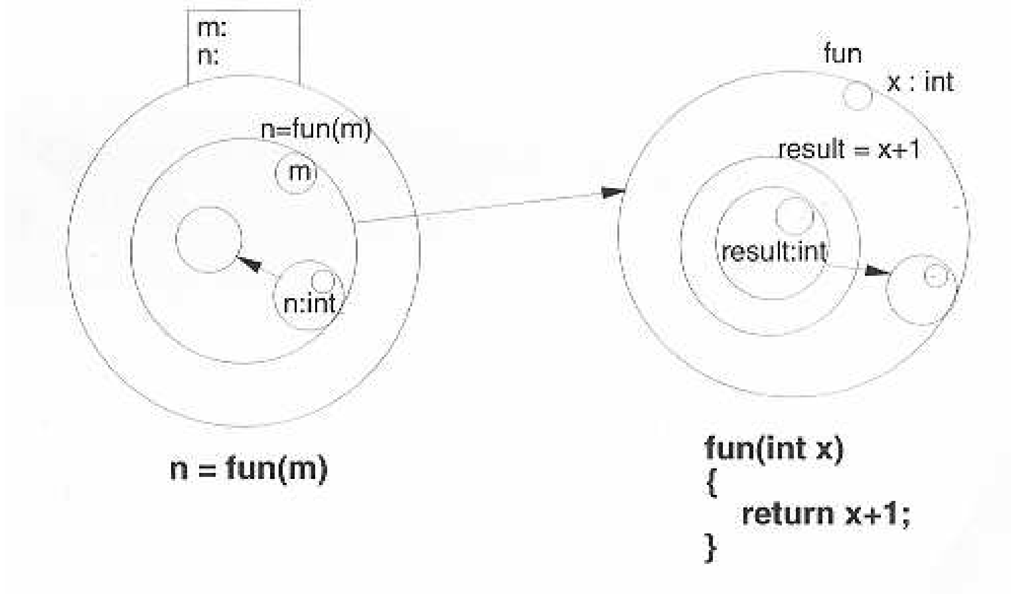


Figure 6: Function definition and call in VIPR

support for both low- and high-level programming.

Figure 5 presents examples of if/then/else and while/do statements in VIPR. Arrows in the figures represent substitution. If execution reaches a ring from which an arrow emanates, then this ring can be replaced with the ring structure to which the arrow points. This idea of substitution becomes more important in function calls, as shown in Figure 6. The left side of the figure represents a call to the function defined on the right. Small circles internal and tangent to rings indicate parameters, so m is a parameter in the call to fun. Every function must have at least one parameter, called the return address or continuation parameter. This parameter indicates the next statement to execute following the function call. The continuation parameter is always located in the lower right corner of the function definition and function call rings. All other parameters can be matched from function call to function definition by either their location with respect to the continuation parameter or by an optional label. For example, in Figure 6 parameter m in the call matches to x in the definition, because they are in the same location relative to the continuation parameter. In the example shown, the small circle inside the continuation parameter ring indicates that the function returns a value. By performing the substitutions called for by the arrows in the figure, we can see that the variable result is passed through to n in the function call.

Figure 7 shows an example of defining a simple geometric point class in VIPR along with the equivalent C++ class definition. The entire class is surrounded by a dotted ring. Private fields or methods are surrounded by double boxes and are not visible in an instance of a class until execution enters a method of the class. Method definitions follow the scheme of function definitions discussed above. Figure 8, Figure 9 and Figure 10 show a small example program which makes use of the point class. Note that variables which have been declared to be pointers to a particular class but have not yet been initialized point to shaded instances of the class. These are referred to as pseudo-instances [Citrin et al. 1994]. The instances become unshaded after the variables are initialized. The only other important aspect of the example to note is the appearance of the self variable in the state upon entry into the "xDistance" method. Clearly, in a program

```
class point {
    int x, y;
 public:
    void moveTo(int newx, int newy)
      { x = newx; y = newy; }

    int xDistance(point* p)
      {
        // Statement 3
        return p->x - x;
      }
};
```

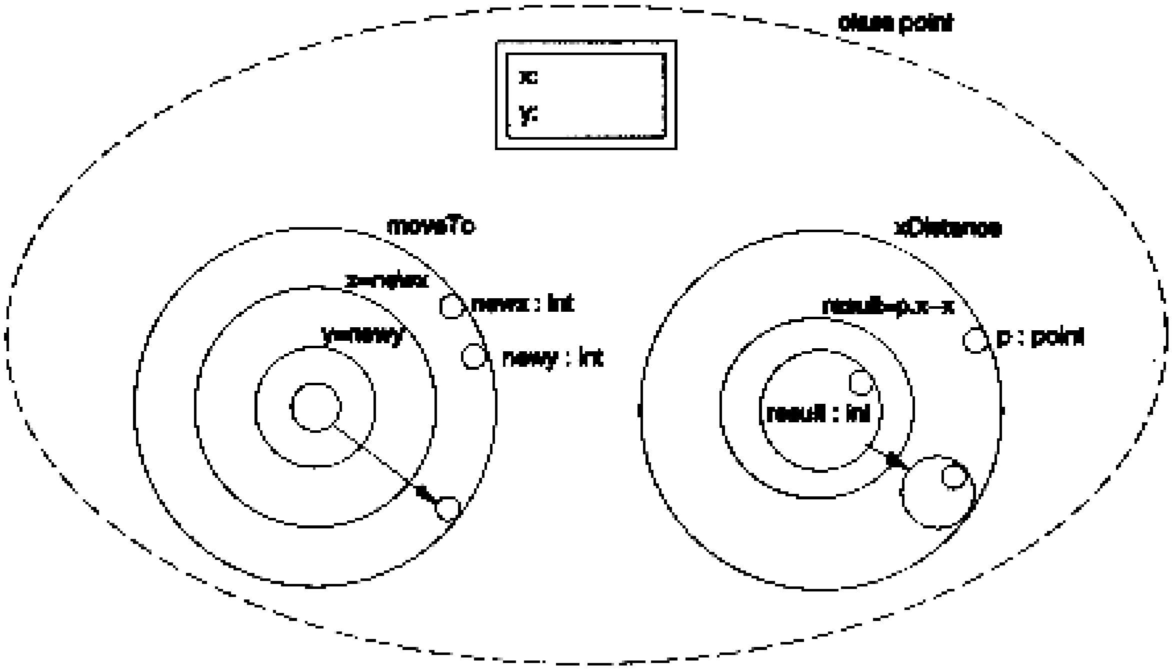
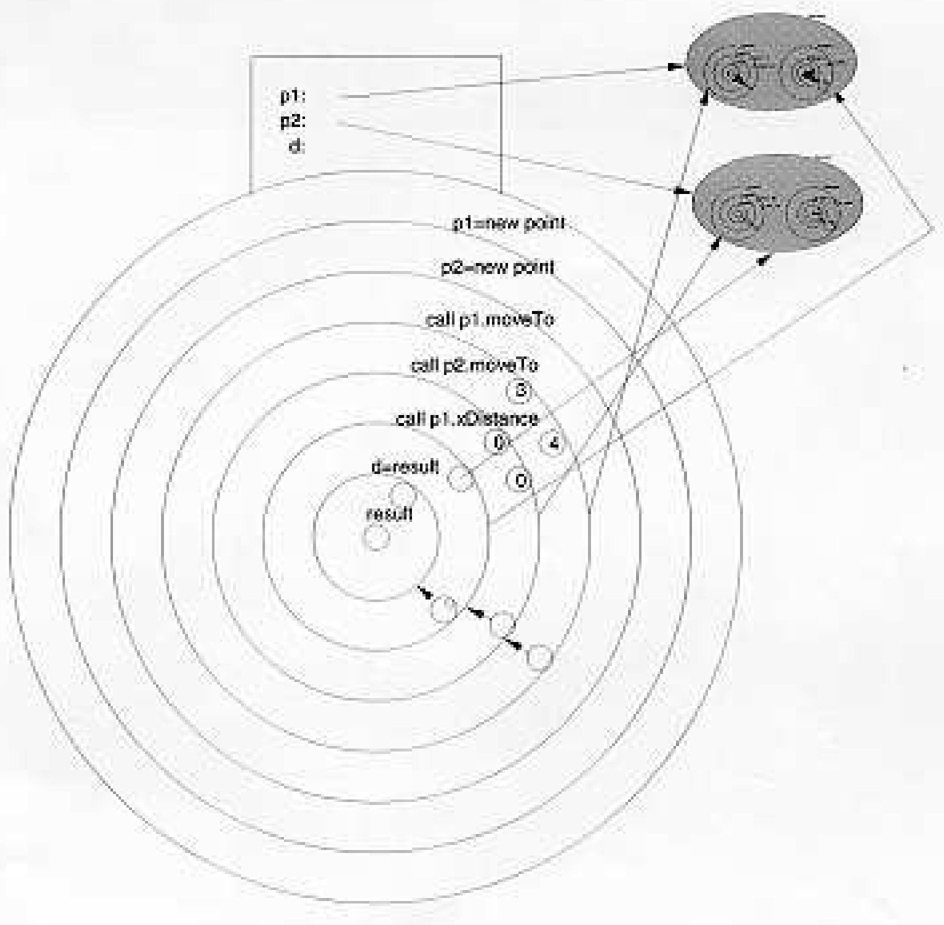


Figure 7: Definition of a point class in VIPR and the equivalent C++ class

```
main()
{
    point *p1, *p2;
    int d;
                              // Statement 1
    p1 = new point;
    p2 = new point;
    p1->moveTo(3,4);
    p2->moveTo(0,0);
                              // Statement 2
    d = p1->xDistance(p2);
                              // Statement 4
}
```



VIPR Program Using the Point Class (at Statement 1)

Figure 8: Example program using point class

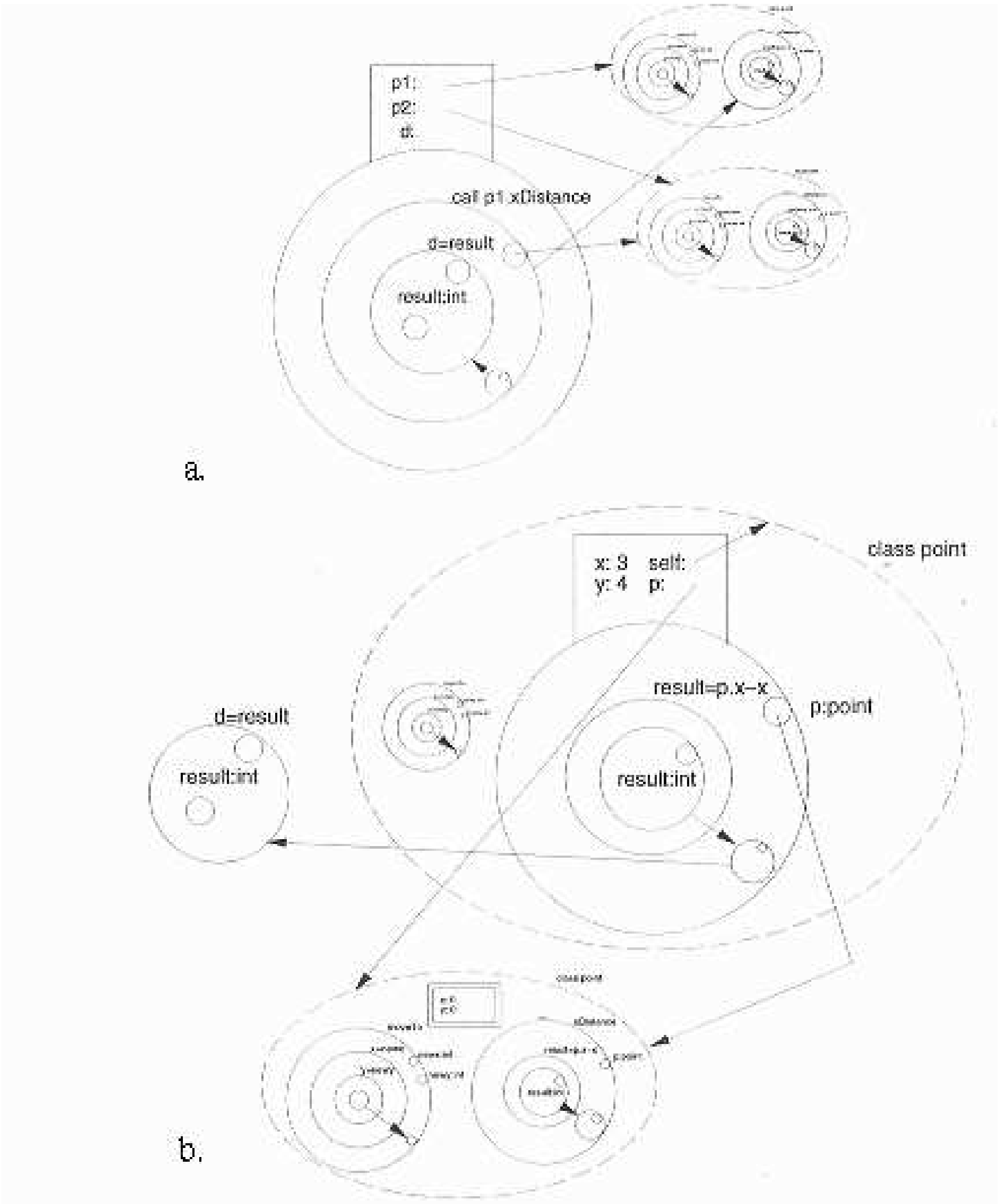


Figure 9: Example program using point class (Cont.) a. at Statement 2 b. inside xDistance

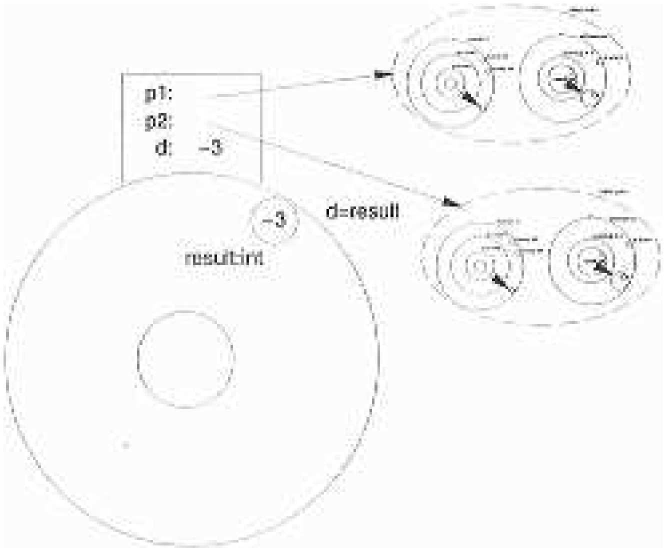


Figure 10: Example program using point class (Cont.) at Statement 4

with a fairly large number of classes and subroutines, the display could become rather crowded. In order to improve program visualization for large-scale projects, Citrin et. al have begun work on a variety of new display methods for the VIPR environment, including zooming and fisheyeing [Citrin et al. 1996].

The VIPR group has also developed a visual representation for the lambda calculus which they refer to as VEX for Visual EXpressions. VEX is intended to become an expression-oriented component of VIPR [Citrin et al. 1995]. We will only take a brief look at its major features. Figure 11 shows the textual and visual representations for the Y combinator. As in VIPR, parameters are represented by small circles inside and tangent to main rings, so f and x are parameters in the example. Function application is represented by adjacent closed figures, and arrows point from the applied functions to their argument. In VEX, free and bound identifiers are easily recognized. Each identifier is connected by an undirected edge to a labeled root node. Free identifiers are connected to roots which are not inside and tangent to any rings, while bound identifiers are connected to internally tangent roots. Thus, in Figure 12 identifier 2 is free in the overall expression while identifier 5 is bound inside the expression represented by ring 3. Graphical equivalents have been devised for α-conversion, β-reduction, and η-reduction, but a detailed discussion of these is beyond the scope of this paper.

6.3 Prograph

In this section we describe the Prograph language which is considered to be the most (commercially) successful of the general-purpose visual languages [Cox & Pietryzkowsky 1990].

The research on Prograph originated in 1982 at the Technical University of Nova Scotia. Since then, several versions of the language have been released, with the most recent (Prograph/CPX) being commercialized by Pictorius, Inc.

Prograph is a visual object-oriented language. It combines the familiar notions of classes and objects with a powerful visual dataflow specification mechanism. Prograph is an imperative

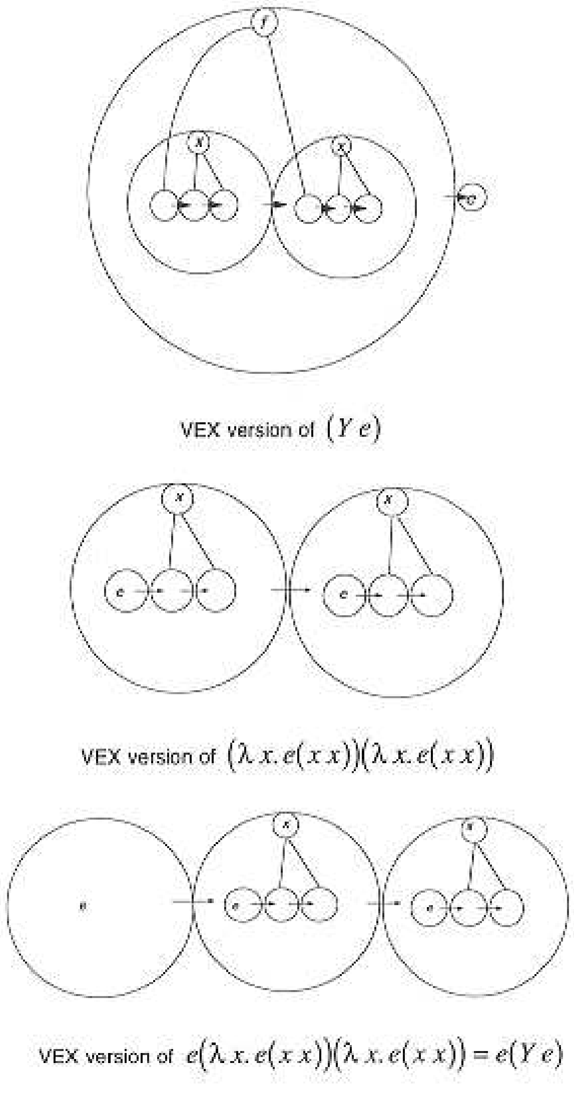


Figure 11: Y combinator $((Y e) = (\lambda x.e(xx))(\lambda x.e(xx)) = e((\lambda x.e(xx))(\lambda x.e(xx))) = e(Y e))$ expressed textually in VEX

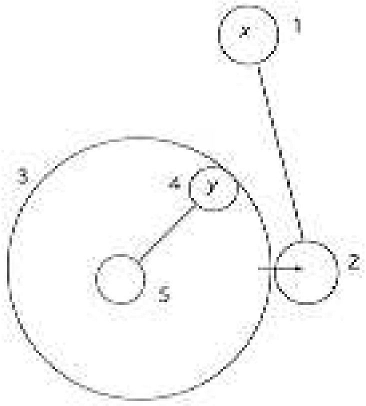


Figure 12: VEX syntax

language, providing explicit control over evaluation order. Of particular interest are Prograph's cases and multiplexes, the special control structures which are intended to replace explicit iteration and provide sophisticated flow control. We will discuss these as part of an example below.

Prograph allows the programmer to work on both high and low levels, allowing him or her to design and maintain more or less complicated software. Primitive computations such as arithmetic operators, method calls, etc. are combined to form method bodies by dataflow diagrams. The methods are then organized into classes, which are, in turn, organized into class hierarchies. In addition, Prograph provides the programmer with so-called persistent objects that can be stored in a Prograph database between different invocations of the program.

We will now introduce some of the more interesting Prograph features by means of an example of topological sort algorithm on directed graphs. Graphs will be represented by adjacency lists, i.e. a graph is a list of lists, each of which corresponds to a node. A list representing a node consists of the name of the node, followed by names of all nodes at the heads of outer edges of this node.

Figure 13 depicts a Prograph program to perform topological sort. The "Methods" window contains the names of all functions in this program, with "CALL" being the top-level routine [1].

Methods consist of one or mores *cases*. Each case of a method is a dataflow diagram that specifies how the case is to be executed. Each dataflow primitive designates an operation to be performed on data which enters through one or more *terminals* (at the top of the operation icon). The output of an operation is produced at one or more *roots* (at the bottom of the operation icon). The edges of a flow diagram indicate how data propagates from the root of one operation to the terminals of the next, i.e. the order of execution is data-driven.

Method "CALL" (also shown on Figure 13) consists of two *cases*. The case number, as well as the total number of cases, is indicated in the title bar of the window containing the method's dataflow diagram. It contains calls to two system methods "ask" and "show" (underlined method names designate system methods) and a call to a user-defined method "sort" whose definition is

[1] All functions in Prograph are referred to as "methods", even those that are not member functions – so-called *universal methods*.

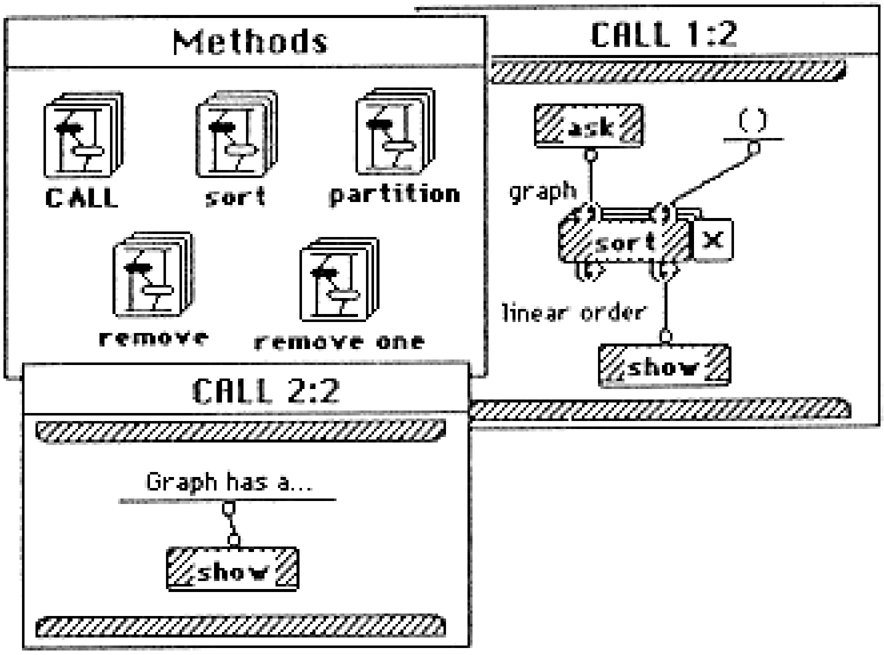


Figure 13: Prograph example – A topological sort algorithm

presented on Figure 14.

The execution of "CALL" begins by evaluating the result of calling method "ask" and the constant "()" – the empty list. "ask" obtains input from the user which becomes the value of its root. After the evaluation of "ask" and the constant is competed, their results are passed to "sort" which is the next operation to be executed. Note that the call to "sort" is accompanied by a control called *next-on-failure* represented by an icon to the left of the operation icon. This indicates that if call to "sort" *fails* (which implies that the graph contains a cycle; see below for the explanation on how this *failure* condition is generated), execution of case 1 of "CALL" should halt and case 2 should be executed.

In case 1 of "CALL", the operation sort is a *multiplex* (which is pictorially distinguished a simple operation by being drawn in a three-dimensional fashion to emphasize the fact that a it is executed many times). At least one of the roots or the terminals of the multiplex is annotated with an icon that indicates the type of the multiplex. The multiplex in Figure 13, is called an *iterative multiplex* (a different kind of multiplex is presented when we discuss the method "sort"). The annotations on this multiplex indicate that output values on the loop roots will be passed to the loop terminals as input values for the next iteration. The execution of the loop continues until a special control called *terminate-on-success* is not executed inside method "sort".

Method "sort" (Figure 14) contains two controls that terminate the execution of the method. The first such control (comparing "sort" input to "()"), called terminate-on-success, indicates to the callee of "sort" that multiplex is to be terminated with "success" status. The second control (comparing output of partition operation to "()"), called terminate-and-fail-on-success, indicates to the callee that the multiplex is to be terminated with "fail" status. As discussed above, this has the effect of failing the execution of the callee and passing control to its next case (case 2 of

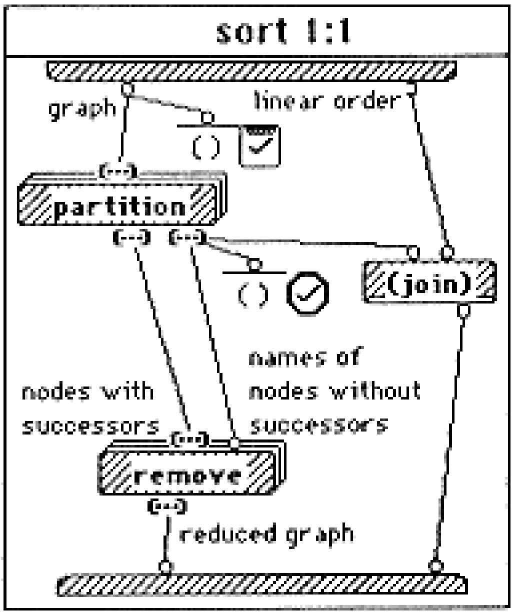


Figure 14: Prograph example – Method "sort" for topological sort algorithm

"CALL" in our example).

Of particular interest is a different kind of multiplex contained in the method "sort", called a *parallel multiplex*. These include calls to "partition" and "remove". This multiplex means that an operation should be applied to each element of an input list (similar to "map" in Lisp) *in parallel*.

Execution of any operation, including method call, can result in any of the following outcomes: execution succeeds, execution fails, execution results in an error. The differentiation of "failures" from "errors" distinguishes Prograph from other case-based languages (such as Prolog) by providing a finer control mechanism. In the case of an error, execution of the program is halted. In other cases, execution continues, according to a control primitive attached to the operation in the body of the callee. This control primitive, marked either by a check or by a cross to indicate whether is it executed on success or on failure respectively, always halts execution of current case and diverts thread of control in one of the following ways:

- Start execution of the next case.
- Indicate failure of the method within which it is contained.
- End execution of calling multiplex.
- Indicate failure in the execution of calling multiplex.

While this brief example is insufficient to illustrate all the features of Prograph, such as its object-oriented model and its interactive programming environment, it demonstrates the essentials of the language. As we conclude our discussion of Prograph, we should mention that the foremost achievement of this language is its ability to free the programmer from tedious chores with the unnecessary level of detail provided by conventional languages, which, in part, is responsible for the language's commercial success.

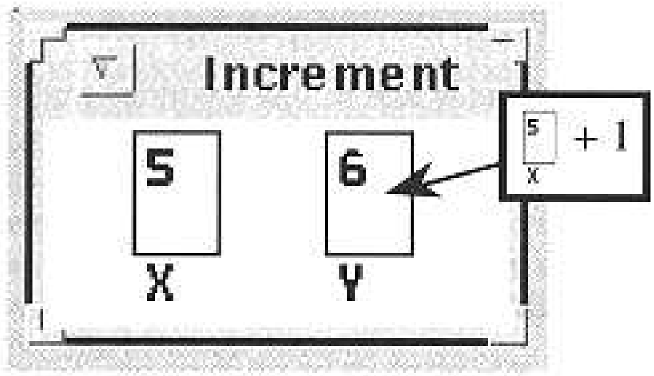


Figure 15: Forms/3: Sample Form

6.4 Forms/3

Forms/3 [Burnett 1994] is another general-purpose object-oriented visual programming language whose features emphasize data abstraction. However, unlike Prograph (Section 6.3) and VIPR (Section 6.2), no inheritance or explicit message-passing is supported.

Forms/3 borrows the spreadsheet metaphor of cells and formulas to represent data and computation respectively. A particular features of Forms/3 is that cells may be organized into a group called *form*, a basic data abstraction mechanism. A form may be given pictorial representation (an icon) and it may be instantiated into an object. In a sense, a form corresponds to a prototype object in prototype-based object oriented languages.

In Forms/3 data (values) and computation (formulas) are tightly coupled. Every object resides in a cell and is defined declaratively using a formula. Objects can only be created via formulas and each formula produces an object as a result of its evaluation. Formulas provide a facility to request results from other objects and create new objects: there's no explicit message passing.

The programmer creates a new Forms/3 program by creating a new form, adding cells to it, and specifying the formulas. A sample form is depicted in Figure 15. The formulas for this form were specified by first selecting cell "X", typing "5", selecting cell "Y", clicking on "X" and typing "+ 1". The programmer could have also referred to cell "X" by typing its name, rather than selecting it on the screen.

Forms/3 implements a declarative approach to flow control combined with the time dimension in an approach that the authors call "vectors in time". With this approach, each vector defines a sequence of objects that represent the value of that cell at different points in time. Returning to the sample form in Figure 15, if X defines a time vector of numeric objects such as <1 2 3 4 5>, then Y defines a time vector <2 3 4 5>. Forms/3 provides the programmer with explicit access to the time dimension, and so iteration can be implemented very elegantly even with this declarative approach. Consider an example of Figure 16, a form designed to compute n^{th} Fibonacci number. Here, "earlier" is one of the time-based operations in Forms/3.

Just as in other general-purpose visual programming languages, Forms/3 allows the programmer to work on both low and high levels. Low-level programming in Forms/3 is performed via formulas, while higher-level abstraction is realized by collecting cells into forms.

In conclusion of our discussion of Forms/3 it is worth mentioning that unlike some visual

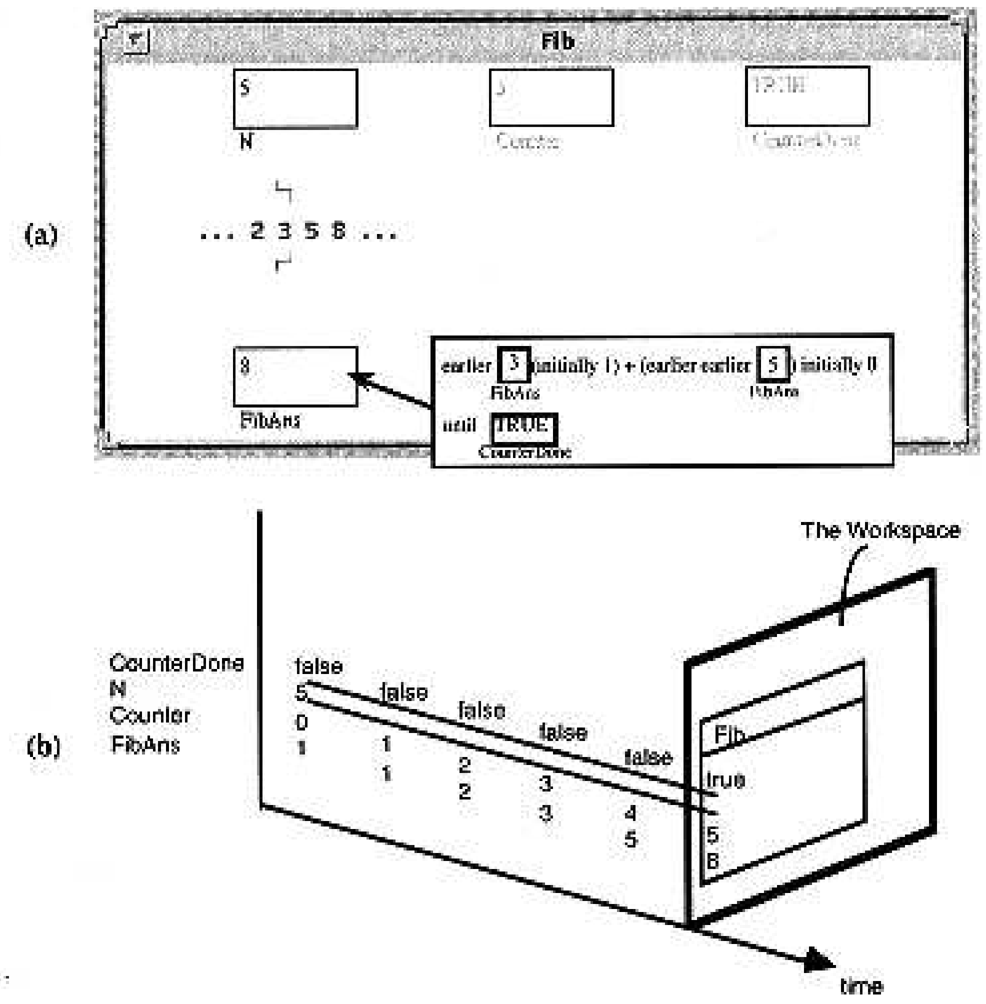


Figure 16: Forms/3 – computing Fibonacci numbers. (a) A form and a formula for one of the cells, and (b) a conceptual sketch of workspace in time

programming language, Forms/3 does not aim to eliminate text completely: the presence of text in formulas is a feature of the language. The objective of the language is to use visual techniques such as direct manipulation and continuous visual feedback to enhance the process of programming.

6.5 Cube

Cube, by M. Najork, represents an important advance in the design of visual programming languages in that it is the first three dimensional VPL. Since Cube programs are translated into simpler internal representations for type checking and interpreting, the language would fall into the category of a hybrid according to our taxonomy. However, the user is never exposed to any textual representation, so the argument could be made that Cube comes very close to being a completely visual language. The language uses a dataflow metaphor for program construction. Working in 3D provides a number of important benefits over more traditional 2D VPLs. For example, working in three dimensions allows the system to display more information in an environment which is easier to interact with than a 2D representation which uses the same screen size [Najork & Kaplan 1991]. In the 3D display, the programmer is free to move his or her viewpoint anywhere inside the virtual world in order to look at any particular section of a program from any viewpoint. This sort of flexibility is not available in most 2D VPLs.

Figure 17 shows the main components of a Cube program as they appear in a recursive function to compute the factorial of a given number [Najork 1995]. Cube programs are composed primarily of *holder cubes*, *predicate cubes*, *definition cubes*, *ports*, *pipes*, and *planes*. The entire structure in Figure 17 is surrounded by a definition cube which associates the icon "!" with the function defined within the cube. The definition cube has two ports connected to it, one on the left and on the right. The left-hand port serves as the input while the right-hand port provides output, although ports in Cube are bi-directional, so technically either port can serve either function. Both ports are connected through pipes to holder cubes in the bottom plane of the diagram which represents the base case of the recursion. Note that each plane represents a dataflow diagram. In the case of the bottom plane, the diagram simply supplies default values for the ports and indicates what type of values each port can accept or produce. If the value at the input port is 0, then the bottom plane is active and the value from the right-hand holder cube, i.e. one, flows to the output port. If the input is greater than zero, the greater than predicate in the top plane is satisfied, and one is subtracted from the input by the bottom branch of the upper dataflow diagram. This difference is fed into the recursive call to the factorial function, and the result is multiplied by the original input. The product then flows to the output port. After defining the factorial function within a program, the programmer can then call it by simply connecting a predicate cube labeled by the "!" icon to holder cubes at the two ports.

7 Conclusion

The field of visual programming languages abounds with examples of unique efforts to widen the accessibility and enhance the power of computer programming. Although the various projects discussed above vary in a number of details, particularly the visual metaphor employed and the targeted application domain, they all share the common goal of improving the programming process. In addition, recent research into solidifying the theoretical foundations of visual programming and serious efforts to develop standardized formal classifications for VPLs indicate that the field has begun to reassess itself and mature. Even as the area has grown over the past twenty years, important historical contributions from work such as Sketchpad and Pygmalion have maintained their influence on various VPL designs.

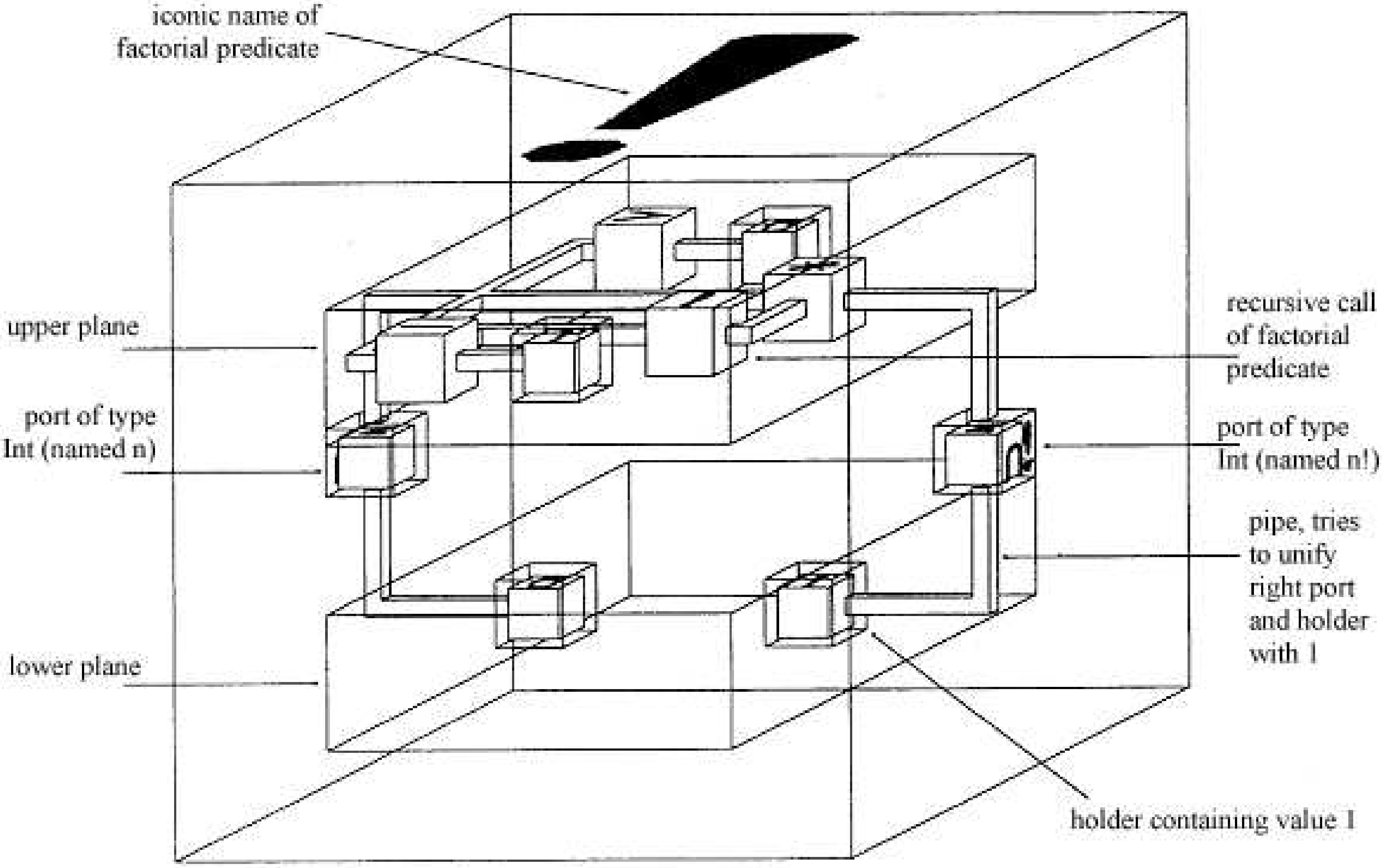


Figure 17: Function to compute the factorial of a number in Cube

Despite the move toward graphical displays and interactions embodied by VPLs, a survey of the field quickly shows that it is not worthwhile to eschew text entirely. While many VPLs could represent all aspects of a program visually, such programs are generally harder to read and work with than those that use text for labels and some atomic operations. For example, although an operation like addition can be represented graphically in VIPR, doing so results in a rather dense, cluttered display. On the other hand, using text to represent such an atomic operation produces a less complicated display without losing the overall visual metaphor.

As computer graphics hardware and processors continue to improve in performance and drop in price, three dimensional VPLs like Cube should begin to garner more attention from the research community. 3D VPLs not only address the problem of fitting large amounts of information on a rather small screen, but they also exemplify the inherent synergy between programming languages, computer graphics, and human-computer interfaces which has been a hallmark of visual programming from its inception.

References

[Borning 1981] Borning, A. H. The programming language aspects of thinglab, a constraint oriented simulation laboratory. *ACM Trans. Programming Languages and Systems*, 3(4):353–387, October 1981.

[Brown & Sedgewick 1984] Brown, M. and Sedgewick, R. A system for algorithm animation. In *Proc. of SIGGRAPH '84*, pp. 177–186, 1984.

[Burnett & Ambler 1992] Burnett, M. M. and Ambler, A. L. A declarative approach to event-handling in visual programming languages. In *Proc. 1993 IEEE Symposium Visual Languages*, pp. 34–40, Seattle, Washington, September 1992.

[Burnett & Baker 1994] Burnett, M. M. and Baker, M. J. A classification system for visual programming languages. *J. Visual Languages and Computing*, pp. 287–300, September 1994.

[Burnett 1994] Burnett, M. M. Seven programming language issues. In Burnett, M. M., Goldberg, A., and Lewis, T. G., editors, *Visual Object-Oriented Programming*. Prentice Hall and Manning, Greenwich, CT, 1994.

[Chang 1987] Chang, S. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, January 1987.

[Chang 1990] Chang, S.-K., editor. *Principles of Visual Programming Systems*. Prentice Hall, New York, 1990.

[Citrin et al. 1994] Citrin, W., Doherty, M., and Zorn, B. Design of a completely visual object-oriented programming language. In Burnett, M., Goldberg, A., and Lewis, T., editors, *Visual Object-Oriented Programming*. Prentice-Hall, New York, 1994. Not published yet.

[Citrin et al. 1995] Citrin, W., Hall, R., and Zorn, B. Programming with visual expressions. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 294–301, 1995.

[Citrin et al. 1996] Citrin, W., Hall, R., and Zorn, B. Addressing the scalability problem in visual programming. In *Proc. of CHI '96*, 1996.

[Cox & Pietryzkowsky 1990] Cox, P. T. and Pietryzkowsky, T. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. In Glinert, E. P., editor, *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Erwig & Meyer 1995] Erwig, M. and Meyer, B. Heterogeneous visual languages : Integrating visual and textual programming. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 318–325, 1995.

[Finzer & Gould 1984] Finzer, W. and Gould, L. Programming by Rehearsal. *BYTE*, 9(6):187–210, June 1984.

[Golin 1990] Golin, E. J. *A method for the specification and parsing of visual languages*. PhD dissertation, Brown University, 1990.

[Lakin 1986] Lakin, F. Spatial parsing for visual languages. In Chang, S.-K., Ichikawa, T., and Ligomenides, P., editors, *Visual Languages*, pp. 35–85. Plenum Press, New York, 1986.

[Najork & Kaplan 1991] Najork, M. and Kaplan, S. The cube language. In *Proc. 1991 IEEE Workshop Visual Languages*, pp. 218–224, Kobe, Japan, 1991.

[Najork 1995] Najork, M. Visual programming in 3-d. *Dr. Dobb's Journal*, 20(12):18–31, December 1995.

[Rekers & Schürr 1995] Rekers, J. and Schürr, A. A graph grammar approach to graphical parsing. In *Proc. 1995 IEEE Symposium Visual Languages*, Darmstadt, Germany, 1995.

[Rhor 1986] Rhor, G. Using visual concepts. In Chang, S.-K., Ichikawa, T., and Ligomenides, P., editors, *Visual Languages*. Plenum Press, New York, 1986.

[Shu 1986] Shu, N. C. *Visual Programming Languages: A Perspective and a Dimensional Analysis*, pp. 11–34. Plenum Press, 1986.

[Smith 1975] Smith, D. C. *PYGMALION: A Creative Programming Environment*. PhD dissertation, Stanford University, 1975.

[Smith 1986] Smith, R. The alternate reality kit : An animated environment for creating interactive simulations. In *Proc. 1986 IEEE Workshop Visual Languages*, pp. 99–106, 1986.

[Smith 1987] Smith, R. B. Experiences with the alternate reality kit: An example of the tension between literalism and magic. *IEEE CG & A*, 7(9):42–50, September 1987.

[Sutherland 1963] Sutherland, I. B. SKETCHPAD, a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pp. 329–346, 1963.

[Tortora 1990] Tortora, G. Structure and interpretation of visual languages. In Chang, S.-K., editor, *Visual Languages and Visual Programming*, pp. 3–30. Plenum Press, New York, 1990.