Out of Flatland: Towards 3–D Visual Programming

Ephraim P. Glinert[†]

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York 12180

E–mail: glinert@cs.rpi.edu

Abstract

The tantalizing potential of so-called visual and iconic programming remains largely unfulfilled because of several unresolved issues. After briefly enumerating the open problems, we review the BLOX methodology which we recently introduced in an effort to overcome some of them. BLOX has been presented in the literature to date in terms of multiple planar diagrams, which impart to the methodology what may be termed a 2.5-dimensional appearance. In the main part of this paper, we show that this is actually but a special, restricted case of the general methodology, which naturally encompasses three dimensions and more. We then argue that it is often unnatural for programmers to compose programs, to view data structures, and to perform other computer-related activities, in less than 3–D.

[†]This work was supported, in part, by the Xerox Corporation.

1 Introduction (Or: From a 1–D Past to a 2.5–D Present)

Over a century ago, the Shakespearean scholar and amateur mathematician Edwin Abbott explored a hypothetical two-dimensional world he called Flatland [1], whose inhabitants were totally unaware of the existence of a three-dimensional Space and the startlingly novel views of the universe that it could provide.

In our view, for technical reasons programming has up to now been similarly artificially confined to just a small part of the programming environment Space.

In the beginning, all programs were one-dimensional (linear) text strings. For many years, these text strings were broken into segments and inscribed on punched cards. Eventually, the medium of expression became glowing phosphor on the glass face of a CRT.

In 1975, David Canfield Smith's dissertation [2] heralded a new era in programming, in which the increased power of computing engines, and their graphics capabilities, made it possible to try to utilize the two-dimensional CRT screen as more than a mere backdrop for the wrap-around of a linear text string. Two new styles of programming resulted. In *visual environments*, multiple windows and graphical elements play

CH2468-7/87/0000/0292$01.00 © 1987 IEEE

prominent roles alongside text (cf. [3,4,5], and commercial products such as the Apple Macintosh human-OS interface and spreadhseet programs). In *iconic environments*, users compose programs by juxtaposing small images, commonly termed icons; in such environments, it is therefore possible to employ well-known "classical" graphical aids to programming such as flowcharts, Nassi-Shneiderman structure diagrams and augmented transition networks, as direct means of human-computer communication (cf. [6,7,8]).

The past decade has witnessed the accumulation of an impressive body of evidence, that visual and iconic environments may prove highly beneficial, both to computer-users in general and to programmers in particular. This is because the computer's ability to represent in a visible manner normally abstract and ephemeral aspects of the computing process such as recursion, concurrency and the evolution of data structures through time, can have a remarkable and positive impact on both the productivity of programmers and their degree of satisfaction with the working environment. Thus, it was not unusual for users of the author's PICT system [6], for instance, to make comments such as "[PICT] would be great for ... home computers, [as] people wouldn't need to learn a computer language." Comments such as this are significant not because they are true, for they aren't; PICT users *did* learn a programming language, as well as a simple editor! The important point is that these remarks are indicative of the users' positive frame of mind after coming into contact with the PICT system for the first time.

The successes of the visual and iconic approaches notwithstanding, the open problems are still numerous and substantial:

- What might constitute an appropriate hard science foundation for the field? Among other things, we sorely need:
    1. Formal models that might lead to counterintuitive results being proven.
    2. Good notations, analogous to the Backus-Naur Form [9] for textual languages, for precisely specifying human-computer interfaces and the visual languages that generate them.
- Can visual programming scale up, so that it will be useful for more than the "toy" programs typically written by novices?
- Can we find, for programs and even more so for data structures, suitable and uniform representations of a mixed textual-graphical nature?
- What constitutes a good "graphical vocabulary?"
- Can we define and validate useful metrics for visual and iconic programming? There are two facets to this problem:
    1. Metrics for assessing the relative merits of alternative environments.
    2. Metrics for comparing programs composed in any given, single environment.

The solutions to many of the problems just enumerated will most likely turn out to be mutually dependent upon one another. In a series of recent papers [10,11,12,13,14] we have introduced a new, paradigm-independent methodology of a mixed textual-graphical nature, which we call BLOX, and provided evidence in support of our contention that an integrated approach based upon this methodology may be one way to attain some of the solutions we seek (cf. [15,16] for another approach).

After reviewing the BLOX methodology in the next section, we will propose in section 3 a more radical approach to resolving these same issues. As BLOX has been presented in the literature up to now, and as it has been used in the design of several environments, the methodology gives the impression of being essentially planar. In this paper we will show that this is an illusion, and we will in fact extend the methodology to three dimensions. We will then argue that it is often *un*natural to compose programs, to view data structures, and to perform other computer-related activities, in less than 3–D. Consequently, the question arises: *Has the time not come for programmers to flee their Flatland prison for the freedom of Space?*

2 Review of the BLOX Methodology in 2.5–D

The classical graphical representations for programs that we referred to in passing in the introduction were all originally developed for use with paper and pencil. As a result, they are two-dimensional and static. The diagrams in question often need to be drawn by hand, a tedious chore for those of us who are not expert draftsmen. Because the graphical representation is distinct from the actual program associated with it, there is often little more than a superficial resemblance between the two after debugging is completed.

A suitable interactive human-machine interface can overcome some of these problems, by mechanizing diagram generation and by unifying the graphical representation of a program with the program itself. Nevertheless, it remains difficult to effectively handle large,

complex programs. One tends either to end up with some form or other of the proverbial spaghetti ball, or to find it necessary to mumble "*encapsulate!*" and resort to hand-waving.

Some investigators believe that the two-dimensional nature common to the classical graphical representations is part of the problem. Thus, they have advocated the use of multiple, interconnected planar diagrams, in what we call a 2.5-dimensional approach [17,18,19].

BLOX representations take all of these ideas one step further. In particular, they have been designed from their inception for display by a computer on a VDU, with far-reaching consequences.

Programming environments based on the BLOX methodology are termed BLOX worlds. The elements of these worlds present an external facade patterned after familiar children's toys. For the purposes of this section, and in conformity with the manner in which the methodology has been presented in the literature up to now, the elements are tiles analogous to the pieces of which jigsaw puzzles are composed. However, BLOX tiles are at once real and imaginary, for unlike their counterparts in the physical world they can hide "miniature" or lower level substructures which they encapsulate (note that this implies a 2.5-D approach). BLOX tiles are also dynamic rather than static, in that their visible features (e.g., size, color, and edge contour or shape) may all change under appropriate circumstances - say, when elements are repositioned on the screen (location/context induced change), or as sundry events transpire (temporally induced change).

Users compose BLOX programs by building structures which consist of one or more joined tiles, according to the usual jigsaw-puzzle lock and key metaphor in which protrusions are plugged into correspondingly shaped indentations so that the two juxtaposed tiles interlock. Depending upon the applications domain, the designer of a BLOX world can, if appropriate, impose additional constraints on which tiles or blocks may be joined. For example, the colors on the interlocking edges or, more generally, the images on the tiles might have to be compatible in some sense.

To elucidate some of the aforementioned concepts, we briefly discuss two examples. In each case, just a few salient features will be described.

Consider first a Proc-BLOX world for programming according to the imperative procedural paradigm. In this environment we would have available tiles which correspond to constructs such as the clause or BLOCK (a sequence of one or more statements, often enclosed, in textual environments, between reserved words akin to the Pascal BEGIN and END), and the IF and WHILE conditionals. However, we would most likely not wish to apply the BLOX mechanism to details at too low a level (e.g.,

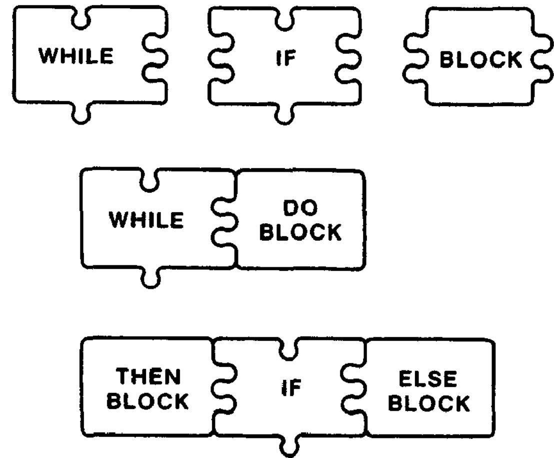


Figure 1: Possible Realizations of Some Imperative Procedural Constructs Using Tiles.

individual characters in identifiers, or the symbols that denote the various arithmetic operations). Thus expressions, say, in assignment statements would be typed in via the conventional keyboard and parsed incrementally (by an interpreter) to assure syntactic correctness.

Fig. 1 illustrates how the aforementioned tiles might look. The actual configurations of knobs and sockets assigned to the various tiles in the Figure are clearly arbitrary and, as a consequence, immaterial. Note how

```
begin
   S1;
   if not I1 then
      begin
         S2;  S3;
      end
   else
      while L1 do
      begin
         if I2 then S4 else S5;  S6;
      end;
   while L2 do S7;
   S8;
end.
```

Figure 2: Schematic Fragment of a Pascal Program.

the lock and key metaphor incorporated in the design of the tiles is used to effectively enforce proper syntax. Note, also, how the shapes of the various instances of the BLOCK tile have been dynamically tailored by the environment to the context in which they are being used. We remark that, in an actual program, the code segments corresponding to the alternative branches, loop bodies and Boolean expressions associated with the IF and WHILE statements could be individually encapsu-

lated in the appropriate tiles, thereby fostering top-down program design. To see how this might work, consider the Pascal program fragment in Fig. 2; Fig. 3 shows the corresponding Proc-BLOX representation. Note that in Fig. 3, the subuniverses contained in some of the tiles have been exploded and encircled for the reader's benefit, although they would not normally all be visible at one time to the Proc-BLOX user.

Let us turn now to an entirely different domain, namely that of VLSI and ULSI design. This is a complex task that commands attention at a multitude of different conceptual levels, including that of complete processors, processor modules as logical units, various gates for the specification of individual logic circuits, and transistors of diverse types, not to mention power sources, signals and wires of multiple types. A VLSI-BLOX world would therefore need to provide access to several families of tiles, along with appropriate tools for working with them. The tiles which denote wires would be particularly interesting, as their shapes would undoubtedly be of a piecewise rubber band nature so that they could stretch and turn corners at will, thereby allowing a wire to link arbitrarily positioned components. These tiles would also probably possess variable translucency, so as to allow sharp differentiation between insulated crossings and junctions. Alternative and sometimes incompatible technologies associated with the physical realizations of certain electronic units could be easily accommodated,

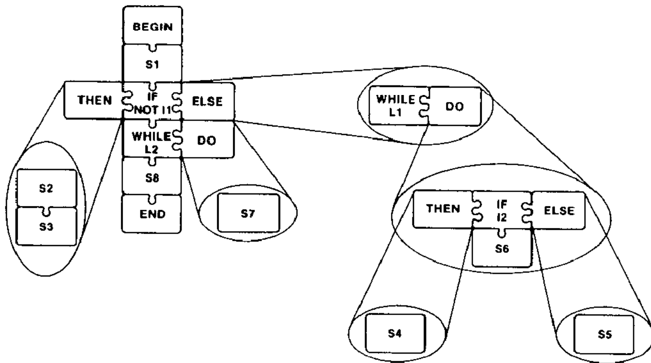
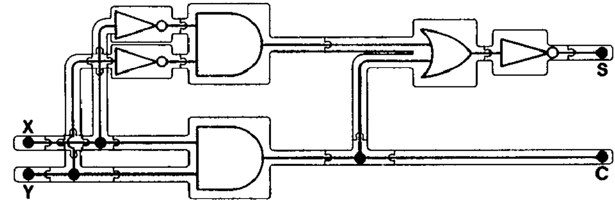


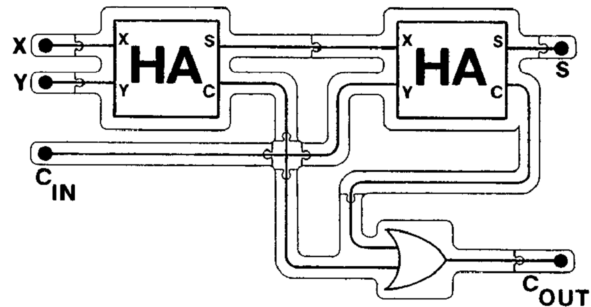
Figure 3: Tile Representation of the Program Fragment Shown in Fig. 2.

through augmentation of the lock and key metaphor, if necessary, with appropriate background colors.

Fig. 4 shows two simple circuits constructed from tiles associated with the gate level of our hypothetical VLSI-BLOX world. We are quick to stress that this Figure is intended to convey to the reader an underlying concept, rather than to depict a design that one might realistically generate on the screen using the environment at hand. Thus, their appearance in the Figure notwithstanding, the edges of the tiles could be made in-



(*a*)



(*b*)

Figure 4: Possible VLSI-BLOX Realization of Two Simple Combinational Circuits: (*a*) Half Adder; (*b*) Full Adder.

visible after insertion into a circuit, so as not to clutter up the image with unnecessary and potentially confusing details. Furthermore, the simple gate level diagrams shown notwithstanding, we are of course aware that floorplans at the higher functional levels, on the one hand, and standard cell or metal layout, on the other hand, would in most instances be the true targets of a VLSI-BLOX world.

As the examples make clear, the BLOX methodology is independent of the underlying programming paradigm, and it also supports embedding of the various classical graphical representations for programs. To see this, note that:

- Proc-BLOX adheres to the imperative procedural paradigm, whereas VLSI-BLOX may be viewed as either a constraint based environment or one that is functional with a graphical front end.
- Proc-BLOX is not flowchart based, because a program's logical structure and flow of control are expressed in a uniform manner by a combination of (*1*) joining tiles together in appropriate ways, and (*2*) encapsulation. Yet BLOX can support flowchart representations, for one can easily imagine an analog to the VLSI-BLOX world in

which gates and wires are replaced by conventional flowchart nodes and flow-of-control paths, respectively.

Having mentioned color in passing several times, it behooves us to direct a few words specifically to the potential role of this attribute in a BLOX world. Although some might argue that color is not essential to the success of a human-computer interface, we fervently believe that its judicious use can prove highly beneficial. Of course, highlighting springs to mind as one possible application, but that's not really the important issue. In complex environments, it is crucial that a hierarchy or vocabulary of graphical clues [20] be provided to facilitate user selection of pairs of tiles that may be joined. A simple hierarchy of this type, in which color supplements the information imparted by shape, might include (in descending order): (*a*) background color or image; (*b*) edge color or pattern; and (*c*) edge contour. This latter attribute could be used to designate subfamilies of compatible elements through use of a master key concept. The arc length of appropriate edges would first be subdivided (by the system designer), after which one or more segments of the contour would be varied while maintaining the remainder identical for all members of the family.

The preceding discussion has centered on two hypothetical examples. Detailed descriptions of the (completed) PC-TILES environment for imperative procedural programming in the small, and of the BLOX-based multiparadigm human-computer interface to OOCADE, an integrated system-level design environment for complex circuits (implementation in progress), may be found elsewhere [21,22].

To complement practical applications for BLOX such as these, we have recently proposed the *class-instance pair*, or CLIP, model for visual programming environments [23,24]. We have shown that BLOX provides a natural embodiment for the CLIP model, which is equally well suited to sequential, parallel and distributed environments. Furthermore, CLIP can support all three major categories of interfaces for programming environments: (*a*) the traditional textual; (*b*) the mixed textual-graphical, as employed in classical graphical representations for programs such as those referred to previously; and (*c*) the highly graphical, as exemplified by systems such as Borning's remarkable ThingLab [25]. The proof for the textual case (*a*) is based on Knuth's well known "boxes and glue" model [26, chapters 11-12].

3 Into the Future: From 2.5–D to 3–D and More

The last paragraph of the preceding section implies, that a textual 1–D BLOX methodology can be obtained as a degenerate version of the 2.5–D case we have described. In this section we will show the converse: BLOX is unique in comparison to the classical graphical representations for programs, in that the 2.5–D version is but a special, restricted instance of the general case, which naturally encompasses three dimensions and more. (About a decade ago, the English researcher R. W. Witty attempted to extend the concept of a flowchart to three dimensions via his so-called "dimensional flowcharting" [27]. However, his ideas were unfortunately ahead of the technology of the time.) The discussion in this section will also clarify our reasons for naming our BLOX methodology with an acronym derived as a synthesis of the word *blocks* with a contraction of *BL*ack b*OX*es.

But first, why do we advocate programming in three dimensions? Many readers will surely argue (and rightly so, as we pointed out in the introduction), that we don't yet know how to properly utilize two dimensions!

We do not propose eschewing 2–D visual and iconic programming for 3–D. We *do* propose broadening our horizons to include the third dimension when appropriate, for several reasons. For one thing, the technology is now available in (top of the line) workstations, and it will rapidly become affordable to all. More importantly, however, are the precedents set by analogy with other branches of science.

There are numerous examples in mathematics, for instance, where extending the domain of discussion allows one to solve seemingly intractable problems relating to the original, restricted domain. Two familiar examples of this are the introduction of fractions to allow division of integers without remainder, and the fact that as a rule the roots of a quadratic polynomial $P(x) = ax^2 + bx + c$ with real coefficients lie in the complex plane.

Furthermore, we repeatedly find cases throughout the natural sciences where it is either actually provable, or generally accepted on the basis of empirical evidence, that 3–D is *significantly different* from 2–D. The following short list of examples should serve to illustrate the pervasiveness and variety of this phenomenon.

1. *In mathematics:*
    - The number of regular polygons, which is finite (and indeed small) in all Euclidean spaces save that which is 2–D, where it is infinite [28].
    - Various phenomena related to random walks, including the probability that the origin will ever be revisited [29].

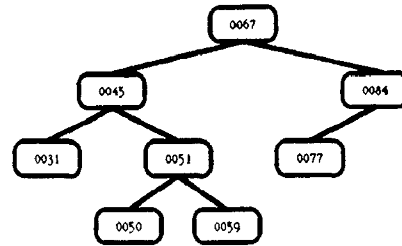
- A host of combinatorics problems such as 3–DIMENSIONAL MATCHING and 3–SATISFIABILITY, which are known to be NP-complete, vs. 2–DIMENSIONAL MATCHING and 2–SATISFIABILITY, which are solvable in polynomial time [30].
- Numerous results in topology, including Borsuk's Theorem, which implies that in Euclidean spaces of odd dimension every continuous mapping of the surface of the unit sphere onto itself (or the negative mapping) has a fixed point, whereas in 2–D a simple rotation about the origin generally does not [31].

2. *In physics:*

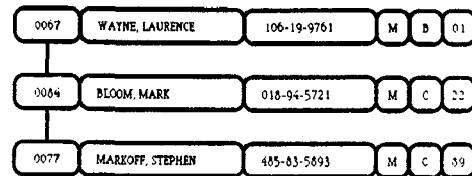
   - Phenomena associated with fluid flow and turbulence, which are often stable in 2–D but unstable in 3–D, or vice versa [32].

3. *In thermodynamics:*

   - The so-called Ising model [33] for describing molecular state transitions has several analytical solutions in the 2–D case, but none are known for 3–D. Moreover, the 2–D solutions imply properties of materials which are often not experimentally confirmed, whereas it is expected that 3–D solutions may better correspond to the real world.



(*a*)



(*b*)

Figure 5: Two–Dimensional Projections of a Tree Structure: (*a*) In the $X-Y$ Plane, Showing the Interelement Links; (*b*) In the $X-Z$ Plane, Showing the Fields Comprising an Element.

Might not a similar situation hold for programming, too? And, if it does, ought we not to exploit it to our advantage?

Consider, for example, a tree each of whose elements is a complex record. In a 3–D space the record fields can be turned at an angle of 90° to the interelement links. Proper choice of vantage point then allows us to view either the overall structure or the contents of certain elements, as illustrated in Figs. 5(*a*) and 5(*b*), respectively. Clearly, for data structures such as this it would be very useful to have a true 3–D display that could be rapidly and continuously rotated in any direction at the user's command.

Because the majority of existent programming languages were designed with 2–D considerations in mind, this domain does not readily provide examples as satisfying as those relating to data structures. (We intentionally say 2–D here rather than 1–D, despite the purely textual nature of most languages, because for more than two decades the choice of constructs provided has usually been determined, at least in part, by the tenets of so-called "structured programming" – a concept which to many refers essentially to the avoidance of spaghetti balls in flowcharts.)

Nevertheless, it is clear that some of the emerging languages, such as the DoD's Ada-based VHDL [34] for designing electronic circuits, are already problematic. Why? VHDL provides concurrent as well as sequential statements, in an attempt to deal with parallelism in complex entities such as CPUs and arrays of memory cells. Upon a first encounter with the language it is immediately obvious that text (1–D) is insufficient to the task at hand; schematic (graphical) capture of designs is imperative in this context [35]. But 2–D or even 2.5–D will at best provide a temporary solution, because the languages of the near future will surely need to address ever more exotic domains: distributed computing, say, or 3–D chip design. As Reiss has recently reiterated [16], in the next generation of programming environments programmers should be able to build programs by putting together collections of objects to yield an executable prototype, so that designers can experiment directly with a design as they work on it.

Even the sequential statements in VHDL are unconventional (and, in some cases, controversial [36]) when compared, say, to those of Pascal. The WHILE, for example, is augmented with both NEXT and EXIT capabilities which violate traditional views of structured programming.

The conclusion is inescapable. It is time for computer science to begin exploring revolutionary rather than evolutionary means of programming, in the hope that the tools will be ready when required.

How, then, can we extend BLOX to 3–D? Actually,

as it turns out this can be done quite naturally; indeed, it is almost child's play. In the simple, 2.5–D form which we described in the previous section, BLOX elements were tiles analogous to the pieces of which jigsaw puzzles are composed. In 3–D, these elements would instead be opaque blocks much like those found in LEGO[‡] or Bristle Blocks.[§] As before, users would compose programs by building structures consisting of one or more elements joined according to the lock and key metaphor, in which knobs are plugged into correspondingly shaped sockets. Again, depending upon the applications domain, the designer of a BLOX world could, if appropriate, impose additional constraints on which blocks may be joined; for example, the colors or, more generally, the images on the touching faces might have to be compatible in some sense.

We noted earlier that, unlike their counterparts in the physical world, BLOX elements can encapsulate lower level substructures. In 2–D this analogy is slightly strained. In 3–D, however, it is completely natural, since we live in a three-dimensional universe; indeed, it is analogous to what toddler's toys such as the Child Guidance Kitty-In-The-Kegs do.

Is our approach to defining the syntax of elements in 3–D BLOX worlds respectable, based as it is on children's toys? We claim that the answer is a resounding "*yes!*" To cite but a single precedent from another realm of scientific endeavor, Linus Pauling's renowned ball-and-stick models for complex molecules are an obvious extension of the Child Guidance Tinkertoy.[¶] Note that the specification and elucidation of molecular structure is yet another domain where a 3–D perspective (i.e., the ball-and-stick model) has turned out to be a valuable supplement to traditional 1–D and 2–D methods (in this case, textual formulas and symbolic planar diagrams, respectively).

Let us close this section by speculating on some of the unorthodox things the elements of 3–D BLOX worlds might do (these may or may not eventually prove to have some "use"). The semantic routine associated with a block might in part depend upon geometric or spatial information such as the color(s) of one or more of the block's faces, the orientation of the block in space, or the immediate neighbors to which it is joined and their relative positions. In addition, as we have pointed out elsewhere [23], a block's semantic routine could trip triggers associated with other block(s) in a variety of ways (triggering can be thought of, in its simplest form, as activation in some sense); the blocks so affected might be immediate neighbors (local), or located at a distance (global). Unusual actions which might be incorporated into a BLOX world could include: cause motion (e.g., rotate part or all of a structure); change the configuration of knobs and sockets, or the image(s), on one or more faces of some block(s); cause the "spontaneous generation" of new blocks and structures, or the replication or deletion of (groups of) blocks.

‡**LEGO** is a trademark of Interlego, A.G.

§**Bristle Blocks** is a trademark of Playskool, Inc., a Hasbro Bradley company.

¶**Child Guidance, Kitty–In–The–Kegs** and **Tinkertoy** are all trademarks of CBS Toys, a division of CBS, Inc.

4 Summary

Expansion of the programmer's work space to three dimensions may make it possible to design new (visual) languages and programming environments, which will differ radically from those in use today and be free of certain drawbacks that plague us at present.

The use of three dimensions for programming cannot be considered an automatic panacea. There are potential pitfalls to be avoided. In the BLOX methodology, for instance, the set of constructs comprising some future programming language might inadvertently be designed so as to allow one block to become completely surrounded by others under certain circumstances. This could make it difficult for users to access parts of their programs with an editor, or to watch execution at run time. (A possible solution to this particular problem could be to make the blocks translucent to a degree.) Nevertheless, we believe that our BLOX methodology, which is naturally extensible from 1–D to 2.5–D, and then in turn to 3–D and more, may prove a useful tool in the design and implementation of these brave new worlds.

Acknowledgments

The author thanks Joseph E. Flaherty, W. Randolph Franklin, Mark Goldberg, David McIntyre, Edwin H. Rogers, and Thomas Spencer for many helpful comments.

References

[1] E. A. Abbott. *Flatland – A Romance of Many Dimensions.* Dover, New York, NY, 1952.

[2] D. C. Smith. *PYGMALION: A Creative Programming Environment.* PhD thesis, Dept. of Computer Science, Stanford University (Technical Report STAN-CS-75-499), 1975.

[3] W. Teitelman. A Display Oriented Programmer's Assistant. *Int. J. Man-Machine Studies*, 11(2):157–187, March 1979.

[4] H. Lieberman and C. Hewitt. A Session with TINKER: Interleaving Program Testing with Program Design. In

Conf. Record of the 1980 LISP Conference, pages 90–99, Stanford University, Stanford, CA, August 25-27 1980.

[5] S. P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Trans. on Software Engineering*, SE-11(3):276–285, March 1985.

[6] E. P. Glinert and S. L. Tanimoto. PICT: An Interactive, Graphical Programming Environment. *IEEE Computer*, 17(11):7–25, November 1984.

[7] B. E. J. Clark and S. K. Robinson. A Graphically Interacting Program Monitor. *Computer Journal*, 26(3):235–238, August 1983.

[8] R. J. K. Jacob. A State Transition Diagram Language for Visual Programming. *IEEE Computer*, 18(8):51–59, August 1985.

[9] P. Naur (editor). Report on the Algorithmic Language Algol 60. *CACM*, 3(5):299–314, May 1960.

[10] E. P. Glinert. Towards "Second Generation" Interactive, Graphical Programming Environments. In *Proc. 2nd IEEE Computer Society Workshop on Visual Languages, Dallas*, pages 61–70, June 25-27 1986.

[11] E. P. Glinert. Interactive, Graphical Programming Environments: Six Open Problems and a Possible Partial Solution. In *Proc. COMPSAC '86*, pages 408–410, IEEE Computer Society Press, 1986.

[12] E. P. Glinert, J. Gonczarowski, and C. D. Smith. An Integrated Approach to Solving Visual Programming's Problems. In *Proc. 2nd Int. Conf. on Human-Computer Interaction, Honolulu*, August 10-15 1987.

[13] E. P. Glinert. *Towards Software Metrics for Visual Programming.* Technical Report 87-16, Dept. of Computer Science, Rensselaer Polytechnic Institute, 1987.

[14] E. P. Glinert and C. D. Smith. Generalized Halstead Metrics for Iconic Programming? In *Proc. Workshop on Visual Programming Languages, Linkoping*, August 19-21 1987.

[15] S. P. Reiss, E. J. Golin, and R. V. Rubin. Prototyping Visual Languages with the GARDEN System. In *Proc. 2nd IEEE Computer Society Workshop on Visual Languages, Dallas*, pages 81–90, June 25-27 1986.

[16] S. P. Reiss. Working in GARDEN: An Environment for Conceptual Programming. To appear in *IEEE Software.*

[17] M. E. Dickover, C. L. McGowan, and D. T. Ross. Software Design using SADT. In *Proc. ACM Annual Conference, Seattle*, pages 125–133, October 17-19 1977.

[18] D. T. Ross. Applications and Extensions of SADT. *IEEE Computer*, 18(4):25–34, April 1985.

[19] M. P. Stovsky and B. W. Weide. STILE: A Graphical Design and Development Environment. In *Proc. COMPCON '87*, pages 247–250, IEEE Computer Society Press, 1987.

[20] G. F. McCleary Jr. An Effective Graphic "Vocabulary". *IEEE Computer Graphics and Applications*, 3(2):46–53, March/April 1983.

[21] E. P. Glinert and C. D. Smith. *PC-TILES: A Visual Programming Environment for Personal Computers Based on the BLOX Methodology.* Technical Report 86-21, Dept. of Computer Science, Rensselaer Polytechnic Institute, 1986.

[22] M.-J. Chung, E. P. Glinert, M. Hardwick, E. H. Rogers, and K. Rose. Toward an Object-Oriented Iconic Environment for Computer Assisted VLSI Design. In *Proc. 2nd Israel Conf. on Computer Systems and Software Engineering, Tel Aviv*, May 6-7 1987.

[23] E. P. Glinert and J. Gonczarowski. A (Formal) Model for (Iconic) Programming Environments. In *Proc. INTERACT'87, 2nd IFIP Conf. on Human-Computer Interaction, Stuttgart*, September 1-4 1987.

[24] E. P. Glinert and J. Gonczarowski. Applications of the CLIP Model for (Iconic) Programming Environments. To appear.

[25] A. Borning. The Programming Language Aspects of ThingLab, a Constraint Oriented Simulation Laboratory. *ACM ToPLAS*, 3(4):353–387, October 1981.

[26] D. E. Knuth. *The TeXbook.* Addison Wesley, Reading, MA, 1983.

[27] R. W. Witty. Dimensional Flowcharting. *Software – Practice and Experience*, 7:553–584, 1977.

[28] H. S. M. Coxeter. *Regular Polytopes.* Macmillan, New York, NY, 1963.

[29] F. L. Spitzer. *Principles of Random Walk.* Van Nostrand, Princeton, NJ, 1964.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, 1979.

[31] J. Dugundji. *Topology.* Allyn and Bacon, Boston, MA, 1966.

[32] G. K. Batchelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, Cambridge, UK, 1967.

[33] H. E. Stanley. *Introduction to Phase Transitions and Critical Phenomena.* Oxford University Press, New York, NY, 1971.

[34] Anonymous. *VHDL Language Reference Manual, v. 7.2.* Intermetrics, Inc., Bethesda, MD, 1985.

[35] K. Bakalar. Oral communication during the VHDL Training Seminar held at UTMC Inc., Colorado Springs, CO, May 18–22 1987.

[36] W. Van Snyder. Multilevel EXIT and CYCLE aren't so bad. *ACM SIGPLAN Notices*, 22(5):20–22, May 1987.