Making Visual Languages More Effective

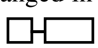
C. M. Holt

Dept of Computing Science, U of Newcastle upon Tyne, NE1 7RU, UK
chris.holt@ncl.ac.uk http://www.cs.ncl.ac.uk/chris.holt/

Abstract

The history of visual language development seems to be full of unfulfilled promise; good ideas are seldom taken up by the outside community because benefits are not perceived to outweigh costs. One reason for this has to do with a certain confusion about how best to take advantage of the flexibility that non-textual syntactic structures should provide. It is suggested that several canonical views of a program be available, reflecting people's various preferences for abbreviation. This is considered within the framework of a dataflow-based notation, but the general points raised should be applicable to other styles of visual language.

1 Introduction

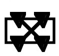
Text can be defined as linear sequences of characters from an alphabet, organized into sequences of symbols where each symbol comprises a connected sequence of characters. Although very useful, powerful, and mathematically tractable, text is often not the best way to convey information. One variation is to use extra-linear positioning: conventions involving integral signs illustrate that two-dimensional information can be used to good advantage. Another possibility is to extend the alphabet to include icons, where these may contain sub-iconic primitives arranged in particular patterns: for example, the icon  (consisting of a square, a line, and a rectangle) can represent a mapping that combines a value with a sequence to yield a new sequence. Also, additional attributes may be attached to linguistic elements: letters and icons may be coloured, italicized, emboldened, etc. to convey particular meanings. Finally, the general richness of geometric structures may be utilized, with lines, squares, circles, cubes, spheres etc. linking and organizing symbols and icons in spaces of one, two and three dimensions. This last can be referred to as programming within virtual reality [6,13].

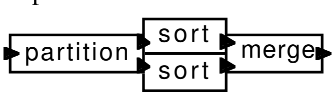
It would seem that if syntax is being enriched, it should become easier to say some things without losing the ability to express oneself clearly in the old ways. Unfortunately, some visual language designers see their task as replacing rather than augmenting text; they want to design an entire new level of syntactic expression, relegating text to a lower, hidden level that might be useful for formal underpinnings, but which would not be utilized by high-level program developers. As with textual programming languages, it appears that completely hiding the sub-structure can be a trap. C became popular as a systems programming language because it let people get close to the machine; visual languages that allow ordinary text to be embedded allow users to take advantage of the centuries of optimization that went into the design of conventional mathematical notations, as expressed in conventional programming languages. However, even when text is mixed with "non-text", problems arise that hinder the spread of visual linguistic conventions.

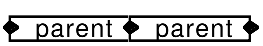
One such has to do with the notion of measuring syntactic complexity [7]; where two languages are used to describe the same algorithm, it can be difficult to determine which description is the simpler. People count characters, symbols, nodes in the parse tree (looking at the abstract syntax), and so forth. They introduce some semantics into their measurements (as with McCabe's metric and function points); but even with all of this, the lowest common denominator is still often lines of code. Of course, such a measure has little meaning when code is not arranged in lines, as generally happens with visual languages; and translating such code into a linear structure necessarily loses the benefits of the original non-linear approach. However, the difficulty of finding a useful metric is not the main reason people approach visual languages with great interest, and then turn away.

A widely perceived problem with visual languages is that of "physical" size: visual programs tend to be big compared to their textual equivalents in terms of sheer space required. If four lines of text convert into a full page of pictures, serious programmers will prefer the former simply because they can see more of a program at once and get a better idea of context and so meaning (barring the extremes of APL-like conciseness). It might seem that this is inevitable given that explanatory examples must use notations unfamiliar to the naive user: the more abbreviations introduced, the more difficult it is to see what is going on. Other textual languages have managed to overcome such hurdles, though; people are willing to learn new paradigms. The proposal here is that various levels of abbreviation be introduced, with simple and explicit rules of translation defined among them. Then, an expert can develop programs quickly in a small space, but such programs can be understood by relative novices by expanding them out into their "long-hand" equivalents. The language viz [x,y] is used as an illustrative example for this process; it is outlined in Section 2, with special focus on its non-traditional features. Section 3 describes the abbreviation levels proposed here, and Section 4 traces through two examples. Even if the details are not directly applicable to other visual languages, it is hoped that the basic approach might be of use in enhancing their potential popularity.

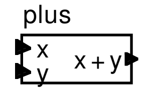
2 An outline of viz

The "language" viz [3] is a collection of syntactic and semantic conventions bound into an underlying framework. The basic idea is that the usual "lines and boxes" dataflow diagram is modified such that interface icons are associated with the objects (boxes) being connected, rather than with the links (lines) that connect them, facilitating a modular viewpoint. An arrowhead icon ▶ indicates an input when it points into a box, while it indicates an output when it points out of a box. Lines indicate links, such that their endpoints have the same values. For example, the K combinator λx,y.x can be represented as , a box with two inputs, one output, and a link between the first input and the output. A feature of this kind of box notation is that it is very easy to denote mappings that have multiple results without having to use tuples; for instance, a Switch mapping λx,y.(y,x) can be represented as . This has an effect on style: composition of mappings in a non-tree structure becomes more natural. Thus, the recursive part of the mergesort algorithm which partitions its argument, sorts each half, and merges the result can be represented directly as

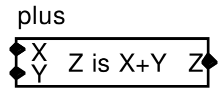


, without needing auxiliary variables to refer to the results of the partition. Boxes can represent relations as well as mappings; the diamond icon ◆ indicates a relational interface in the Prolog sense. Thus, the composition of two relations indicating parenthood 

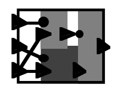
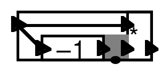
yields a relation indicating grand-parenthood. A box may have both input/output interfaces and relational interfaces; this mixing tends to be clumsy to describe using text alone.

A box may contain a name or a structure, where a name is an icon or identifier. A name placed by a box is given that value; a name placed by an interface is associated with a link such that every use of the name indicates an endpoint of the link. For instance, the (trivial) Miranda definition "plus x y = x+y" can be directly translated into 

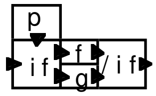
(though of course this can be simplified by eliminating the link names "x" and "y"). The Prolog equivalent "plus(X, Y, Z) :- Z is X+Y" is directly translatable into

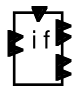


(this too can easily be simplified). The question of when a name beside an interface defines a new link (as opposed to when it represents a value that has already been defined) is left to the conventions of the visual space containing the box, and is not part of the viz framework (though the default is that what an instance of a name refers to is an attribute that may be resolved by looking at the "map" of the name, a hypergraph structure relating positions within and between visual spaces).

The circle icon • indicates a link to the contents of a box. This facilitates higher-order operations; for instance, the S combinator λx,y,z.(xz)(yz) can be represented by  (using shading to indicate internal boxes). It can also be used to indicate recursion, when an internal box is linked to the one containing it; e.g. the recursive part of a Miranda definition of factorial (i.e. "fact n = n*fact(n-1)") can be represented by 

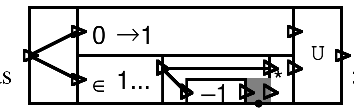
(where "−1" means decrement).

The dataflow view encourages analogies to current moving along wires, or light packets moving down optical fibres. This in turn makes it natural to think of propositions succeeding or failing in a Snobol-like way (though less operationally): if a condition succeeds, it returns a value; if it fails, it returns a special value Fail (rather like ⊥). [This is the total-mapping approach, as opposed to that of partial-mappings in which there is no explicit Fail value.] The if-then-else operation "if" takes two arguments: a value and a proposition to be applied to the value. There are two outputs: if the proposition holds, the value is returned through the first, and the second returns Fail; while if the proposition does not hold, the value is returned through the second, and the first returns Fail. There is a mutex operation "/if" that returns the first of its arguments that does not Fail; most operations are strict w.r.t. Fail, so after an if-then-else, failure generally propagates until such a mutex is reached, in parallel with the successful branch. That is, the standard "if p(x) then f(x) else g(x)" is replaced by 

, where f and g are evaluated concurrently (and an optimizer omits the intermediate steps needed for failure propagation). There might be more than one argument to a proposition p (e.g. for such things as testing for equality between two arguments, or determining whether one argument is greater than another); then, it is natural to think of p as succeeding if the desired property holds, and returning all its arguments if so (while returning Fail at each of its results otherwise); this can be indicated by having multiple inputs and outputs, e.g. 

can be used for a conditional that accepts pairs of arguments and returns pairs of results. It should be noted that a value passing through such a structure can be a mapping from names to values (i.e. a state in the imperative style); the mappings f and g here can be state transformers, e.g. assignments like "x := x+5" which yield new states.

One of the principles behind viz is that any operation a person can perform on a program should itself be executable as a program; thus, programs can construct, modify and run other programs, because the editing commands available to a person (creating and destroying objects and links, accessing their values and components, and changing these) are part of the language. This means that there may be program components that are lacking inputs because they have not yet been connected up, but which have to be given a well-defined semantics. In such instances, it would be possible to define the output to be Fail; however, it is useful to distinguish between an operation that is missing an input from one that has an unacceptable input, since the former can be refined purely by adding extra conditions, while the latter requires non-monotonicity to model the transition. For this reason, an Undecided value has been introduced; if an input is missing, the output is (usually) Undecided. Most operations are strict w.r.t. this value; however, there is a truth operation that returns true if its argument is a value (neither Fail nor Undecided), false if its argument is Fail, and intermediate if its argument is Undecided. That is, there are three truth values, which are distinct from the arguments that give rise to them; although truth(Fail)=false and truth(Undecided)=intermediate, truth(false)=true and also truth(intermediate)=true. This enables a program to reason about the status of other programs; in this sense, the truth operator is in the meta-language, and is reflective.

Values can be superposed on one another such that operators act on them independently, roughly in the way that Scott defined operators on sets of values in [12] except that superposition is not idempotent; it corresponds to a bag-union rather than a set-union, such that $x \cup x = 2x$. $\cup$ satisfies the equations $f(x \cup y)=f(x) \cup f(y)$ and $(f \cup g)(x)=f(x) \cup g(x)$, and can be used to join different parts of a function or relation definition; Fail is the identity element for $\cup$. It serves the same purpose as the (implicit) operation combining multiple rules in Prolog or Miranda, except that because of its commutativity guards need to be applied to each branch. For instance, factorial can be represented as 

: if the argument is 0 the result is 1, while if the argument is 1 or more, the recursive expression is called. If the argument is the superposition of more than one number, the result is the superposition of evaluating factorial for each of them. There is a unite operator that combines all the components of a superposition into a single value; as with the truth operator, this is needed to reason about the status of programs and is in the (reflective) meta-language.

The notion of time is present in viz; when an operator is connected to an input, it is evaluated and the output changes from Undecided to whatever the result is. The value at the output is a time sequence that undergoes transitions, in this case from Undecided to the result. If the input is then disconnected and a different input connected, the output time sequence reflects this by changing to Undecided and then to the new result. Operators that involve time sequences include separating them into heads and tails (and the adjoint of this), and mapping them to lists (and the adjoint of this). Time sequences involving superpositions of values are equivalent to superpositions of time sequences. It can be useful for an operator to "hold on" to an argument after it has been disconnected; for instance, a dyadic operator that accepts one argument which goes away before the other arrives might otherwise end up not producing a result. The interface icon ▶■ indicates that the operator is allowed to curry itself, such that changes in that input are ignored until all other inputs arrive, at which point it is reset. An operator may have any or all of its inputs stabilized in this way. Another kind of stabilization, represented using the interface icon ▶|, holds on to an argument through Undecided intervals, but releases it when a new argument arrives; then, the time sequence for that argument consists entirely of values and Fails. When this icon is used for an output, any output value persists through an Undecided interval until a new output is calculated. A mapping is temporally characterized by the delay between changes in input values and changes in output values. This delay may be different for different values, of course (as with factorial). There is a wait operation which takes a duration as one argument and delays any transition by the given duration, without changing any values that pass through; this is useful in programs that explicitly manage time.

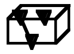
3 Shorthands

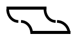
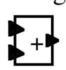
The purpose of a shorthand is to encode information densely, taking advantage of additional knowledge and vocabulary (as in defining new objects as structures of old ones) or context (as in omitting parameters when one of them is constant in a given scope). The purpose of this section is to show how visual shorthands can be systematic, rather than ad hoc; it should not be necessary to know the particulars of a given program fragment to understand how various shorthands can be used within it. This is not a new idea; many programming languages contain features that can be defined in terms of other constructs of the language. Indeed, it was the widespread practice of this kind of language extension that led to notions of a "core" language, whose semantics had to be very precisely defined, together with an "extension" language, defined in terms of the core. For example, while and repeat loops can be defined in terms of recursion; the implication arrow "–>" of Prolog can be defined in terms of cut; and the map of Miranda can be defined in terms of more general higher-order functions.

Visual shorthands are not so well understood. It is easy to appropriate those of textual languages, but less so to take advantage of the general geometric freedom available to construct concise equivalents that are of use in making programming easier. A particular danger is that setting up conventions reduces the freedom of the programmer to tailor notation to the needs of particular applications; every rule imposed by a language designer limits a user. Therefore, a hands-off attitude is taken with respect to: (i) binding particular dimensions in a space to semantic domains, e.g. by representing time on an axis or by defining parameters on one or two dimension and mappings of those parameters on another; and (ii) giving meaning to visual attributes of objects such as colour or minor deformation (analogous to bold or italic characters). This leaves the following ways in which information can be carried: (iii) non-orientation-based positional relations such as adjacency or distance can be bound to relations; and (iv) conventions can be devised for relating structures in spaces of different dimensions. [There are likely to be other ways in which information can be carried; the above is not meant to be exhaustive.]

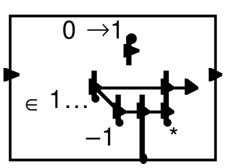
Option (iii) above involves topological properties relating objects in a space. Two open sets can have non-empty intersection, they can have their closures intersect while they do not, or their closures can be disjoint; informally, they can touch, be next to each other, or be distant. It was suggested in [5] that touching might indicate combination into a data structure, though the nature of that structure (i.e. how it combines its components) might best be left open at this level; different touching patterns might represent quite different things. It was also suggested that adjacency represent application, following the textual convention in which a function is placed beside its arguments. Again, local variation seems possible; adjacency in different dimensions might have somewhat different meanings. As a refinement, if this seems too onerous a convention then some apparently adjacent objects could actually be made to touch by giving them transparent envelopes; this convention is often employed with text horizontally on a page (e.g. combining characters into symbols), though not vertically. Finally, it was suggested that distant objects enclosed within the same frame be considered bag-joined, i.e. superposed. Geometric conventions can interfere with this option in that a given dimension can be bound to an ordered domain, such that objects distant in that dimension can be related by that ordering.

More important here is the question of (iv) above, that of translation between different dimensions: is it feasible to translate structures among the dimensions and so simplify the programming task? The two kinds of structures to be considered are objects and the links among them (though links can themselves be viewed as objects that identify their ends). In the general case, it seems clear that when moving from 0D to 1D to 2D to 3D, each dimension adds a degree of freedom (and so expressiveness); when going in the other direction, it is not possible to encode all the information directly. There are two factors that give cause for hope, however. On the one hand, the freedom available as a result of extra dimensionality is often not actually used; on the other hand, information can be stored indirectly, such that it can be acquired by enquiring of an object what its attributes are. An example of this is the way types are associated with ordinary programming variables: once declared, a variable does not display its type, though a decent user interface would make such information easily available upon enquiry. Similarly, although properties of an object might be displayed syntactically in higher-dimensional spaces, this information may be hidden (though accessible upon request) in lower-dimensional representations.

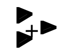
To see what this means in practice, we start with 2D boxes as described in the previous section. It is easy to extend these to 3D, just by giving them depth: a rectangle becomes a rectangular prism. Internal structure and/or interfaces may be given depth or not; the simplest approach is to let them remain 2D unless there is good reason to do otherwise. Thus, the K combinator becomes . Going in the other direction, if it is desired to render a 3D shape in a 2D space, the two obvious possibilities are to take a characteristic slice (or a modification thereof), using that as the 2D shape; or take a snapshot of the 3D image from some angle. In either case, the 3D information is then hidden, to be made accessible upon request or upon translation into a 3D environment once more.

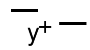
The translation from 2D to 1D follows the same basic principle: a line segment is chosen to represent the 2D structure. A rectangle is most naturally represented as a straight segment, while a shape such as  might be represented by . One issue in going from 2D to 1D involves giving the line segment semantic content; there is no way that a name or structure can be placed within it. This can be resolved using the contents icon •; a link to such an icon either by line or by name is sufficient to determine what it represents. The other issue has to do with interfaces, since there is no inside or outside, left or right which can be used to distinguish them. An order has to be defined over the 2D boundary that places the interfaces in a sequence; they can then be placed on the 1D representation maintaining that order. For instance, the dyadic mapping 

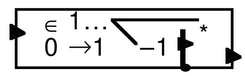
can be ordered such that interfaces on the left precede those on the right, so its 1D equivalent is 

, with the first two arrowheads accepting inputs from the left, and the third emitting output to the right. A further shorthand can be introduced, by overloading inputs and outputs, such that the output can emerge to the right from one of the input arrowheads; this is not feasible in the general case, but occurs often enough to be useful. The factorial function can be rewritten using these 1D conventions, yielding 

(this uses the convention that distance implies superposition between the 0→1 case and the recursive case, and the multiplication overloads the first arrowhead to be both input and output).

The function lines in this example feel awkward; they seem to add clutter without adding much information. This is the motivation behind looking at the possibility of translating from 1D to 0D, where the latter means that a name represents a point that corresponds to that name. This may also be understood as taking a 2D object containing a name, and removing the boundary, since a name necessarily occupies 2D space. This loses any information carried by the shape of that boundary, though as before it can be stored as an attribute. Interfaces can be recognized by their position relative to the name, where that is taken as central; for example, the "+" mapping above can be represented by 

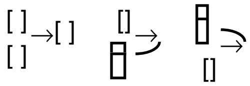
. As a further shorthand, interface icons can be omitted when they can be inferred from position; the structure 

has an explicit link coming into its first input, the link named y coming into its second, and the output goes to the explicit link on the right. Incorporating these conventions, factorial becomes 

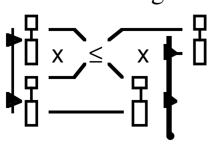
(keeping the line for the recursive call and making the inputs and outputs for the two parts implicit). Of course, if a reader finds this too terse or difficult to understand, it can be expanded in part or in whole back up to a higher-dimensional representation, or the overall size can be enlarged to emphasize the "gaps" between components, and so focus attention on the implicit connections.

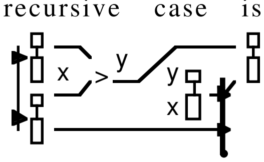
The above has concentrated on the dimensionality of objects, but that of links is important as well. A 0D link is one that has been given a name, such that each instance of the name represents an endpoint of the link; this is the usual textual convention. The standard lines-and-boxes diagram replaces some or all of these named links with explicit lines; an example is the link between the output of "∈ 1..." and the inputs of both "–1" and "*" in the above factorial example. 2D links may be used to carry lists of values among objects; if continuous links are allowed, they may carry mappings from continuous linear domains to values. 3D links may similarly carry bijective mappings, where each dimension of the cross-section of a link is associated with one of the domains of the mapping. [Since these links may carry time sequences, such mappings may be time-variant.] In going from 3D to 2D, or 2D to 1D, a parameter must be hidden. For instance, if a 3D link carries the value 5 at position (8,10), and the dimension carrying the second argument is suppressed, then the corresponding 2D link carries the mapping 10→5 at position 8 (as it carries all the other mappings found at position 8 in the 3D link). If this is to be translated to a 1D link, then that link carries a dyadic mapping such that given the arguments 8 and 10, the result is 5. Thus, each dimension in a link can be bound to a domain just as can happen in the space containing a visual structure.

4 Examples

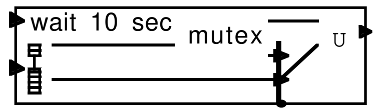
This section offers a couple of examples to show how shorthands can be used when actually developing programs. The first is the standard "sequential" merge function, that compares the heads of its two arguments, chooses the smaller, and merges everything else. It tends to look complicated in ordinary lines-and-boxes languages because of its non-planarity: the spaghetti effect starts coming into play. The first step is to define the base cases, where either or both argument lists are empty: 

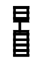
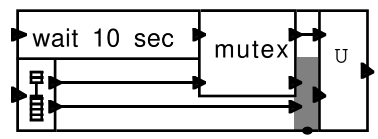
. These components are dyadic mappings. The first indicates that if the arguments are both empty lists, then so is the result. The second indicates that if the second argument is not empty (the icon  indicating that there is a head with a value in it), then that argument is the same as the result. The third does the same for the first argument.

The first recursive case partitions the two argument lists, compares the heads, and if the first head is smaller it is concatenated with the merge of the first tail with the second list: 

. Points to note here include: (i) the arguments have explicit, linked interfaces so it is clear they are together; (ii) the  icon is used for both splitting a list into its head and tail, and also for combining a value and a list into a new list, with the number of parameters indicating which operation is required; (iii) the name "x" carries the tail of the first list, because an explicit link would involve two crossings and so introduce greater syntactic complexity than using the name does; and (iv) a line is used to indicate the recursion, with the output being emitted from the top arrowhead and going into the final concatenation. The second recursive case is not very different: 

. Here, the head of the second list goes to the top, so the head of the first list must go down to be combined with the tail of the first list, before merging the result with the tail of the second list.

A second example involves the use of time sequences; it is a watchdog timer, that is reset if fed a signal within 10 seconds. If it is not fed such a signal, it emits an alarm. One way to construct such a timer is 

. The first input is the starter of this instance of the timer; it is delayed for 10 seconds. The  icon splits a time sequence corresponding to the remaining signals into a head and tail in a way analogous to the operation for lists. The head is the next signal; if it arrives within the required 10 seconds, it blocks out the initial signal, and a recursive instance of the watchdog is started. Otherwise, the initial signal wins, is joined with the output of the recursive call, and an alarm is sent. If an alarm is generated within the recursive call it is output through the current instance; thus, any alarm is propagated upwards (and an optimizer can ensure this takes negligible time). The full 2D box version of this is 

, which is not much larger but which takes more effort to produce.

5 Related Work

There has been considerable work done on multiple views of programs, of which a recent visual example is [8],. However, this tends to focus on presenting different attributes of programs with different views, rather than translating between different kinds of syntactic conventions with the aim of allowing programmers to use hybrids. The various dimensions also have a history; the most recent is 3D (e.g. [1,2,4,11]; [10] involves 3D, but is directed towards program animation rather than design). Again, however, the idea of using different dimensionalities within a program does not often appear. The idea of mixing text and graphics is not new (e.g. [9]); but this does not incorporate both text and higher dimensions within a single view.

References

[1] G Franck and C Ware, Representing Nodes and Arcs in 3D Networks, in: Proc IEEE VL94, St Louis (IEEE CS Press, Los Alamitos Cal, 1994) 189-190.

[2] E Freeman, D Gelernter and S Jagannathan, In Search of a Simple Visual Vocabulary, in: Proc IEEE VL95, Darmstadt Germany (IEEE CS Press, Los Alamitos Cal, 1995) 302-309.

[3] C M Holt, viz: A Visual Language Based on Functions, in: Proc IEEE VL90, Skokie Il (IEEE CS Press, Los Alamitos Cal, 1990) 221-226.

[4] C M Holt, Visualizing an Algebra of Objects, in: Proc IEEE VL91, Kobe Japan (IEEE CS Press, Los Alamitos Cal, 1991) 225-230.

[5] C M Holt, An Algebra of Lines and Boxes, in: Proc IEEE VL94, St Louis (IEEE CS Press, Los Alamitos Cal, 1994) 55-62.

[6] C M Holt, Programming within VR (TR 529, Dept. of Computing Science, U. of Newcastle upon Tyne, Oct 1995).

[7] C M Holt, Comparing Visual and Textual Languages (TR 536, Dept. of Computing Science, U. of Newcastle upon Tyne, Oct 1995).

[8] Y Koike, Y Maeda and Y Koseki, Improving Readability of Iconic Programs with Multiple View Object Representation, in: Proc IEEE VL95, Darmstadt Germany (IEEE CS Press, Los Alamitos Cal, 1995) 37-44.

[9] M Kopache and E P Glinert, C^2: A Mixed Textual/Graphic Environment for C, in: Proc IEEE VL88, Pittsburgh Pa (IEEE CS Press, Los Alamitos Cal, 1988) 231-238.

[10] H Lieberman, A Three-Dimensional Representation For Program Execution, in: IEEE VL89, Rome Italy (IEEE CS Press, Los Alamitos Cal, 1989) 111-116.

[11] M A Najork and S M Kaplan, The CUBE Language, in: Proc IEEE VL91, Kobe Japan (IEEE CS Press, Los Alamitos Cal, 1991) 219-224.

[12] D Scott, Data Types as Lattices (SIAM J Comput 5 3, Sept 1976) 522-587.

[13] VRML 1.0 (http://www.virtpark.com/theme/vrml/