Programming within VR

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

As different kinds of behaviour are added to VRML-based webspaces, a variety of styles, tools, interfaces, and languages are likely to be developed, enabling people to program objects io perform actions of considerable complexity. The WYSIWYG principle will lead to an increase of programming by example; and it will be noted that the requirements for describing and controlling such behaviours are very similar to those needed for real-time concurrent programming in general; this is one of the areas the visual language community is trying to address. One approach to dealing with the Tower-of-Babel effect is to offer a unifying framework within which the various efforts can be situated, enabling them to interface together; this paper offers some suggestions as to what such a framework might consist of.

1 Introduction

The VRML community is beginning to grapple with the problem of introducing behaviour into three-dimensional webspaces (3DW). Mailing lists [24,25,26] have considerable bandwidth devoted to the question, and the relative merits and features of systems like Tbag [22], Java [23], and Spam [21] are being examined. Much of the discussion is engineering-oriented, in the (non-disparaging) sense that people are trying to figure out the best way to get efficient, scalable behaviours that do what people want.

In many ways, this is analogous to the way concurrent languages were developed. People began by looking at existing features used to control parallelism in operating systems, and they generalized and abstracted, creating semaphores, monitors, actors, and so forth. As time progressed, it was seen that reasoning about concurrency required a mathematical model of processes and intercommunication; thus, for instance, Hoare's rather informal CSP programming notations [7] led to the theory of CSP [8]. [Popular alternatives, of course, include CCS and Petri nets.] Similarly, as objects were developed and various refinements explored, it was seen as desirable to put them on a formal footing, and there are various attempts to do this, e.g. based on order-sorted algebras [6].

It seems likely that as various kinds of behaviours at different levels are added to 3DW, there will again be a desire to fit these into a mathematical framework. This would allow them to be expressed precisely, and to be related to one another; and it would also provide a guideline for simplicity, since it often happens that mathematical complexity is an indication that underlying concepts are difficult to learn and use in a scalable way (as with goto's). It is therefore not too early to start considering the nature of the foundations that might be required to describe object behaviours and interactions.

One preliminary observation is that experience obtained in formalizing programming constructs could well be helpful in this undertaking. A program, after all, is simply a description of behaviour (though declarative languages obscure this to the extent that they hide interactions with the environment). Thus, it might be possible to take advantage of the state of the art of programming language semantics, applying it to this particular domain, without reinventing rotating wheels, as it were. Another observation is that the visual language community has been engaged in trying to formalize image-related descriptions and behaviour for some time (most recently e.g. [9,10,12,18,19]), and that lessons can be drawn from that experience. A third place to look for inspiration is directly to mathematics, and the ways that mathematicians have tried to lay a solid foundation for that subject; after all, programming languages can be characterized as notations for describing particular kinds of mathematical relationships, and so a foundation for mathematics would (in this view) necessarily be a foundation for reasoning about mathematical characterizations of behaviours.

The goal is to sketch out an approach to developing a framework within which a wide variety of behavioural models can be embedded; it is not trying to propose a language for describing behaviours, although the framework will of necessity make some languages seem more "natural" than others. In particular, it is concerned with the way that behaviours can be represented within 3DW itself, rather than within some other textual object-description language. The reason for this is that the structures used for text can (and should) be easily embedded within 3DW. They were developed over a long period of time, and have been carefully optimized for various application areas that are common and important (there is no point in writing a "+" on the side of a 3D block if you can just write the "+" on its own to indicate addition). However, linear text on its own can be severely constraining (hence the growing popularity of graph grammars and hypertext). The basic approach is to characterize the semantic domain, in the sense of what behaviour means; to relate that to abstract syntactic domains, both source (what is being said) and target (what it is being said about); and then to relate these to rather more concrete syntactic entities, that represent the abstract syntax.

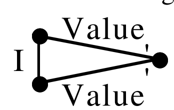
The paper begins by looking at static systems: it considers the nature of the semantic domain of visual entities, how such things can be represented linguistically, and what the relationship is between these. Then, in Section 3, the question of time is considered. This involves looking at the additional mathematical apparatus needed to represent time in the semantic domain (e.g. temporal logics [1,5]), seeing how time can be represented either statically or dynamically, and finally considering temporal scoping questions.

2 Static Spaces

The starting point for constructing a model of visual entities is to select foundational domains. As with mathematics, there are a number of choices, each of which can be used to construct the others. For instance, set theory (be it ZFC, Quine's NF, or whatever) can lay the groundwork for category theory, and vice versa. [The Preface of [13] notes relations among logic, category theory, and type theories.] In either case, essential concepts (such as the role of names or equality) are taken from the meta-language, formalized, and used with precision in describing the entities and structures of the other. As with mathematics, the underlying concept here is that of a point, an entity that is undecomposable. Points have the property that they can be differentiated, one from another.

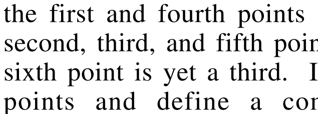
One choice to be made is whether to take relations or mappings as foundational. Either can be used to define the other: a relation can be viewed as a mapping with an implicit boolean result, while a mapping can be viewed as the special case of a relation in which if "result parameters" differ, so do "argument parameters". Relations are chosen here to avoid the need for adjoints, which would otherwise continually arise; the "space" of entities is less cluttered. Thus, there are points, and there are relations among those points. [Going higher-order, there are also relations among relations, relations among these, and so forth [15].] A relation is something which can be said to hold over parameters, where a parameter may be one or more points, and parameters are distinguished one from another.

A relation in which one parameter is a point may be said to define an attribute of that point. For instance, if the relation Colour holds between a point P and another parameter Red, we may speak of P as having the Colour attribute Red. [If there are no other Colour relations involving P as that parameter, we may view Colour as a mapping from P to Red.] We may say that a point can be characterized in terms of its attributes; this means that its significance is derived from its relationships. If a relation is viewed in terms of attributes of the points it relates, it is implicit; if it is viewed as an entity in its own right, it is explicit. One important relation is that of Value; a point has a given value if the Value relation holds between it and exactly one other entity. Two points have the same value if they are both related by Value to the same entity; then, the identity relation I holds between them. This may be illustrated by the commutative diagram



where ' distinguishes the "parameter position" of a point being related. That is, position is significant for parameters of Value, though it is not for I.

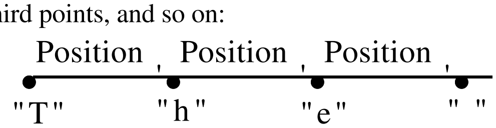
A diagram like the above, in which points are connected by lines, defines a number of compositional rules. When a point is involved in more than one relation, the relations may be composed to form another relation. Thus, the diagram indicates that going from a point to its value, and then to another point that has the same value (i.e. composing Value with its inverse, in the usual sense) is equivalent to going from the first point to the final point via I. Relational composition is defined to be associative; when "going along" a relational path containing more than two relations, if those relations are pair-wise composed, the order of composition does not affect the overall result. Note that relational structures as used here must in general be represented by hypergraphs rather than just graphs. An instance of a relation may have more than one point in a given parameter, and it may have more than two distinguished parameters; e.g. in



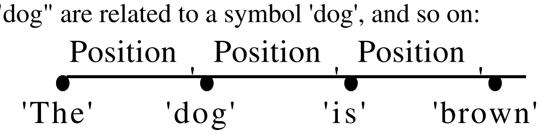
the first and fourth points are one parameter, the second, third, and fifth points are another, and the sixth point is yet a third. If we were to name the points and define a commutative combining operation, this might be represented textually as R({1,4},{2,3,5},{6}).

The mathematical structures that we are working with can be defined as allegories [4]. An allegory is to a category as a relation between sets is to a mapping between sets. Analogizing from the description of a category in [11], it consists of (i) a class of objects (so far merely points, but this will be expanded); (ii) a class of relations, each of which has distinguished domains which are objects of the allegory; and (iii) a composition law over relations which is associative and for which there is an identity element. Allegories may be thought of as containing structure-preserving relations: if an object O1 is related by R to an object O2, such that a point O1.p1 is related to O2.p1 and another point O1.p2 is related to O2.p2, where O1.r relates O1.p1 and O1.p2, then R relates O1.r to a relation O2.r, such that O2.r relates O2.p1 and O2.p2.

One motivation for dealing with allegories arises when dealing with sentence spaces, where a sentence space is a structure belonging to a given language. Such a structure consists of points with attributes and relations, and is related by a Meaning relation to some other structure. When dealing with textual languages, sentence spaces start with a linear structure; for instance, the sentence "The dog is brown." is a linear space containing 17 points. Each point has a Character attribute; the first one being "T", the second being "h", and so on. Linearity is derived from an explicit Position relation, which relates the first and second points, the second and third points, and so on:

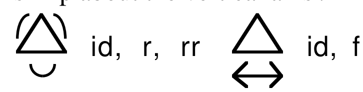


This sentence space is related by Meaning to another sentence space, in which points have Symbol attributes, and are (still) related by a linear Position relation. The first three characters "The" are related to a symbol represented here by 'The', the characters "dog" are related to a symbol 'dog', and so on:



The relative positions of characters in different words are preserved; e.g. "T" precedes "d" in the character space, and its symbol 'The' precedes 'dog' in the symbol space. The symbol space is also a sentence space, since it too is related by Meaning to another space. This latter space may be in the underlying semantic domain, or it may be another intermediate step. For instance, in textual programming languages, linear symbol spaces are generally related to trees according to the BNF grammar rules that define the language syntax; then, the tree spaces are related to semantic objects.

There is no sharp division between syntax and semantics, in this view; a given space may be semantic in that it is the target of one Meaning, and syntactic in that it is the source of another. For instance, a group generated by two values r (of period 3) and f (of period 2) is an entity in its own right, with six elements {id, r, rr, f, fr, frr}; but it can be given a meaning, that of invariant operations on an equilateral triangle. Then, r means 'rotate by 120°' and f means 'flip about the vertical axis':



Of course, a sentence space can have more than one meaning, a given meaning can be represented by more than one sentence, and more than one meaning relation can be introduced.

A space (consisting of points and relations) may itself be viewed as an object in an enclosing space, and so be related to other objects in that enclosing space. One such relation between one structured object and another is that of a View; this is a sub-relation of I (like a retract [17]) that restricts the components of the space that can be seen. An object may have more than one View relation; other objects may be able to see different parts of it. To the extent that an enclosing space may be understood as seeing its components, it too may have a view of objects within it, i.e. parts of those objects may be hidden.

Just as relations may be composed, if they share one or more points, so may objects be composed, by causing them to share components (which may be points, relations or sub-objects). Shared components are unified, in the sense that attributes associated with the components in each of the sharing spaces are present in the result. An operation very similar to sharing is that of identifying components, i.e. introducing an identity relation between different entities. This may be defined such that each entity gains the attributes of the others; it may be one sided, such that some entities are left unchanged while others acquire their attributes; or it may be an intersection, such that each entity is left with only those attributes common to all those being identified. A further refinement is to allow some attributes to be unioned, some left unchanged, and some intersected.

Given these underlying notions of points, relations, spaces and objects, we can start to look at visual languages and representations. The semantics of a programming language can be translated into this structure without much trouble. Prolog-style relational languages are trivial; relations are associated with named points whose values are the relations, and parameters are named points that are linked together. Functional languages follow the usual function to relation translation, with application being a particular relation between function points and their argument points. Imperative languages can be understood as sequences of states related by state transformers, which are mappings, and so are amenable to the functional translation. Object-oriented languages associate objects with those above; methods are interface structures that can be viewed by other objects. Concurrency is an implementation technique, but communication can be via unified points as channels. Complex issues relating to types and inheritance are concerned with the construction of systems of relations and objects, which is not dealt with here; so, although very important, it may be avoided.

A program space may be represented by selecting some of its relations and making them explicit, while making the others implicit. This yields an abstract syntax space, with a structure corresponding to the explicit relations. Names can be introduced to reduce the clutter of explicit relations; when several objects are associated with the same parameter of a relation, they can be linked by a Name relation, which identifies them all with one another. This name can then be viewed as an attribute, and will guide the representation of the objects in question, such that each occurrence of the name will indicate the equivalence class of the points being identified (and so indicate referential transparency, if that name is not used elsewhere).

An abstract syntax space can then be represented in a concrete space by devising representations for the various components. The concrete space may be one, two, or three dimensional; it may contain objects represented in equivalent and lower dimensionality. [E.g., a 3D space might contain named spheres and 2D pages linked together by lines.] Possible representations are drawn from the vocabulary of the concrete medium. Some relations may take advantage of the relations inherent in the concrete medium; for instance, vertical height (from a baseline) may be used to carry information about a particular relation among points, and so make it explicitly visible in the space without cluttering up the representation of the points in question. The concrete space can then be represented in the usual VRML way, as a display list which is rendered as a screen image (or two screen images for binocular vision).

Statement spaces can describe visual scenes if the vocabulary of the display domain is incorporated into the semantic domain of the language; if this is the same as the concrete (or abstract) syntax space of the language, it is syntactically reflective (in the appropriate way). With VRML at present, of course, the language space is textual; if visual representations are introduced, this could change. [One motivation for doing so is to simplify the task of denoting components; one can "quote" them in situ, e.g. refer to a blue sphere by displaying a blue sphere.]

3 Dynamic Spaces

The simplest kind of change that a space can undergo is invisible: it maintains its static structure, as represented in the concrete space, but various attributes are modified. For instance, if a structure represents an expression, with a named point representing the result, then evaluation of that expression will bind the value of the named point to the result value. Since results that cannot be accessed are not very interesting, we look at ways in which spaces can be modified. One principle of general purpose programming language design is that if a person can perform an action, then a process should also be able to; so the issues of human-directed change and machine-directed change are considered together.

The ABCD mnemonic of object design (access, birth, change, and death) lists the things we want to do with spaces; these may be characterized as internal or external, depending on whether the object initiates action or whether some outside influence does. Access involves obtaining information from an object. If this is apparent from its representation, it may be categorized as internal; if it is not so apparent (e.g. finding the value of a result as above), it is external. Birth means that new objects are introduced to spaces, by creating them ab initio, by copying objects (either from within the space or elsewhere), or by merging two or more objects to form a hybrid, being able to select which components of the source objects are unioned, intersected, or just taken from one source rather than another. When a single object is created, some impetus must come from outside of it before it exists; this is external. When objects merge, this can be either external (another agent causes the merge) or internal (the merging objects act on their own). Change can involve devising a transition for an object, creating or deleting components of an object, or modifying the structure relating objects to one another; this can be caused either externally or internally. Death means that objects vanish; it may be triggered internally, such that it kills itself, or externally, such that it dies irrespective of its internal state or actions.

Given this list of possible actions, it is possible to start characterizing the ways they can be described. One approach is to have an explicit temporal structure in which points represent states. If the points are discrete, then an action can be specified by giving the initial state and the nature of the change (e.g. an assignment), and the final state can be deduced. Alternatively, the initial state and final state can be given (as in a temporal logic with a "next" operator [16]), and the change can be determined by a kind of empirical induction. If time points are continuous, differential equations can indicate changes (though, of course, an implementation on a digital machine would discretize these). Hybrid systems containing both discrete and continuous time are possible, of course, obtained by composing the two [14].

Time can either be global (totally ordered), running on a universal clock, or local (partially ordered), running on distributed clocks. In point of fact, a hybrid can be constructed here as well; local clocks can engage in occasional synchronizations, avoiding the heavy-weight monolithic character of universal lock-step time, but allowing objects to refer to time in a common frame of reference (albeit with a precision loss dependent on the synchronization mechanism). Looking at a time map in this latter case, one would see a directed acyclic graph of nodes and arcs, with branch points when new local clocks are created and joins when they merge together. Actions are bound to points and lines in time maps by just another relation.

Time maps describe process objects, fragments of behaviour that are triggered (i) by an explicit call, (ii) by some external condition being changed (that the already extant process has access to), or (iii) by some internal change. Following the Erlang [2] and Oz [20] approaches, these may be described as conditionals; when the trigger event occurs, the associated action is taken. [The trigger may involve clock transitions, since a clock is just another object.] When a number of trigger/action conditionals are composed together, they may be depicted as a graph; when the number of possible states is small, this may look like a state-transition diagram, and involve cycles. Conditionals must be bound to time maps, indicating the temporal scope over which they may be activated; and they may be one-off, serial, or multiple/concurrent.

Communication among objects is a matter of access (seeing what another object is displaying) or change (modifying an available part of another object). The former is asynchronous, in that the behaviour of the "sending" object is not directly affected by the communication. The latter may be viewed as synchronous in that the "sending" object is affected by the "send" at that time, and the "receiving" object is also affected by the action. However, the intent of the action may be to affect a third object, which is accessing the one that is changed (much as two people can use a piece of paper to communicate); then, the communication is synchronous only if the trigger in the third object is directly focussed on the part of the intermediate object being changed, and reacts immediately. If it is not, this kind of communication is asynchronous. Of course, protocols can be built on top of this to implement hand-shaking communication of the kind found in CSP and CCS; and buffers can be used as intermediate objects to implement message queues as found in traditional asynchronous approaches. RPCs and Ada-style rendezvous can also be constructed from the above. [The above can be constructed using handshakes, RPCs, or rendezvous; however, simulating one-way communication using two-way is inherently slower than constructing two-way communication from one way. Thus, the latter, as well as fitting into the framework more neatly, is more efficient, and is therefore to be preferred.]

The discussion so far as focussed on what is to be said, rather than how to say it. When objects and actions can be easily referred to, e.g. by selecting attributes that they alone possess in the current environment, no problem arises. If we wish to add 5 and 3, we can create an object, give it a value associated with the symbol '+', select one of its interfaces and give it a value associated with the symbol '5' (or create a new object and give it that value, and then unify the "output" of 5 with one of the "inputs" of +), do the same with 3, and link the result interface to some display object (either by unifying it with an input to the display object, or by creating an identity relation which is attached to both).

When an object is to be created by copying another one, we either signal to its "replicate" interface, or ask the environment to replicate it (if it is fully transparent in that environment). If it is to be built from scratch, the attributes have to be constructed; but these can be replicated (and then possibly changed). When an object is to be changed from the outside (assuming the changer has permission, i.e. its view is sufficient), a component may be created or destroyed, or one may be replaced by another. [These are standard editing commands, generalized to objects.] When the relationship between an object and its enclosing space is to be changed, then actions are invoked to transform that relationship. This may involve ordinary editing of the space, such as dragging an object; it may involve giving the object an acceleration, which induces a velocity that may be affected by "physical" laws of the space (which include the existence of other objects that may or may not be solid); or it may involve allowing the object itself to modify the physical laws of the space, e.g. by changing its gravity. When changes are "programmed" by doing them, either by a person or a process, then there is no difference between run time and edit time; what's done is done, and no time map is needed. When changes are described (scripted) to be done later, time maps may be left implicit ("do action A at time T" doesn't involve the construction of a map) or they may be constructed explicitly.

4 Conclusions

Once interactive behaviour is introduced into VRML, it will become a language for describing programs (among other things), since programs are simply descriptions of machine behaviour. However, there will be many programming languages, over all the paradigms, that will be given a visual syntax. Rather than favour one over the others, it is suggested here that a foundation can be laid, upon which the various languages can be built, and implemented over different platforms. The simplest such framework seems to have a semantic domain of points and relations, and objects composed of these; the syntactic domain used to describe this is just that of VRML descriptions. [An assumption here is that optimization can give reasonable efficiency when needed.] The construction of visual behaviours then involves giving programming languages access to the syntactic domain, much as various languages now access the Tk widgets [3], and introducing time and event/action conditions. This would enable the VRML community to take advantage of the experiences of programming language design of the past few decades, and offer potential flexibility without constraining it to follow a single path.

References

[1] J Allen, Maintaining Knowledge about Temporal Intervals (CACM 26 11, 1983) 832-843.

[2] J Armstrong, R Virding and M Williams, Concurrent Programming in Erlang (Prentice Hall, New York, 1993).

[3] M J Conway, Python: A GUI Development Tool (Interactions II.2, April 1995) 23-28.

[4] P J Freyd and A Scedrov, Categories, Allegories (North-Holland, Amsterdam, 1990).

[5] D Gabby, Modal and Temporal Logic Programming, in: A Galton, ed., Temporal Logics and their Applications (Academic Press, London, 1987) 197-237.

[6] J Goguen and R Diaconescu, An Oxford Survey of Order Sorted Algebra (Math Struct in CS 4 3, Sept 1994) 363-392.

[7] C A R Hoare, Communicating Sequential Processes (CACM 21 8, Aug 1978) 666-677.

[8] C A R Hoare, Communicating Sequential Processes (Prentice Hall, New York, 1986).

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

[10] F Jahanian and A Mok, ModeChart: A Specification Language for Real-Time Systems (IEEE Trans SoftEng 20 12, Dec 1994) 933-947.

[11] P T Johnstone, Stone Spaces (CU Press, Cambridge, 1982).

[12] G Kutty et al, Visual Tools for Temporal Reasoning, in: Proc IEEE VL93, Bergen, Norway (IEEE CS Press, Los Alamitos Cal, 1993) 152-159.

[13] J Lambek and P J Scott, Introduction to Higher Order Categorical Logic (CU Press, Cambridge, 1986).

[14] Z Manna and A Pnueli, The Temporal Logic of Reactive and Concurrent Systems (Springer-Verlag, Berlin, 1992).

[15] Karl Meinke, Universal algebra in higher types (Theoretical CS 100, 1992) 385-417.

[16] B Moszkowski, Executing Temporal Logic Programs (CU Press, Cambridge, 1986).

[17] J E Stoy, Denotational Semantics (MIT Press, Cambridge MA, 1977).

[18] J Tsai, S Yang and Y-H Chang, Timing Constraint Petri Nets and Their Application to Schedulability Analysis of Real-Time System Specifications (IEEE Trans SoftEng 21 1, Jan 1995) 32-49.

[19] H Vestli, Visual Specification of Actor Configurations, in: Proc IEEE VL94, St Louis (IEEE CS Press, Los Alamitos Cal, 1994) 110-117.

Websites

[20] DFKI, The Oz Programming System (http://ps-www.dfki.uni-sb.de/oz/).

[21] Paul Lalonde, Robert Walker, Jason Harrison and David Forsey, A Coordination Framework for Interacting Agents (http://www.cs.ubc.ca/spider/lalonde/spam/spam.html).

[22] G Schechter et al, The TBag Project (http://www.sun.com/smli/sun-rt/tbag/tbag-home.html).

[23] Sun Microsystems Inc, HotJava (http://java.sun.com/).

Mailing lists

[24] vrml-behaviors@sdsc.edu (http://www.sdsc.edu/SDSC/Partners/vrml/repos_mailing.html).

[25] distributed-vr@csh.rit.edu (http://www.csh.rit.edu/~airwick/dist.html).

[26] www-vrml@wired.com (http://vrml.wired.com/).