Research report 133

EVALUATING DEFINITIVE PRINCIPLES FOR INTERACTION IN GRAPHICS

Meurig Beynon

(RR133)

This paper is an appraisal of current progress towards supporting interactive graphics within the framework of a general-purpose programming paradigm based upon definitions. It considers how the use of definitive principles relates to other work, why it appears promising, and what progress has been made towards the resolution of technical difficulties. As a sequel to [2], it re-examines potential for applications of definitive principles in interactive graphics in the light of more recently developed ideas about dealing with control issues and dynamically changing relationships in a definitive programming framework. It also takes account of new research into notations for graphics that makes use of geometrical constructions. As a subsidiary theme, the paper contrasts the support for reference and representation of geometric relationships in vaious kinds of interactive graphics systems.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

November 1988

Evaluating definitive principles for interaction in graphics

Meurig Beynon
Department of Computer Science
University of Warwick
Coventry CV4 7AL
e-mail: wmb@uk.ac.warwick FAX +44 (203) 525714 Tel: +44 (203) 523089

ABSTRACT

This paper is an appraisal of current progress towards supporting interactive graphics within the framework of a general-purpose programming paradigm based upon definitions. It considers how the use of definitive principles relates to other work, why it appears promising, and what progress has been made towards the resolution of technical difficulties. As a sequel to [2], it re-examines potential for applications of definitive principles in interactive graphics in the light of more recently developed ideas about dealing with control issues and dynamically changing relationships in a definitive programming framework. It also takes account of new research into notations for graphics that makes use of geometrical constructions. As a subsidiary theme, the paper contrasts the support for reference and representation of geometric relationships in various kinds of interactive graphics systems.

Key words: interactive graphics, spreadsheets, geometric constructions, constraint-based graphics systems, functional programming, reference, animation

1. Introduction

Most existing interactive graphics systems focus on providing the user with a toolkit that makes it possible to draw complex diagrams; they reflect a view that is oriented towards tools rather than frameworks. The emphasis is upon programming as a "means to an end", viz the depiction of a complex diagram. Though graphical interfaces are fashionable, and user-computer interaction concerns are deemed important, the underlying idiom often resembles batch programming. The user is an unequal partner who makes use of an interactive interface for convenience, but is not thereby enabled to intervene more significantly and directly in the computational process.

Modern applications for graphics demand a broader perspective. It has become necessary to think of "interaction in the large": to consider the issues of managing many thousands of drawings, perhaps over many years of development, within the broader semantic context of such applications as engineering design and manufacture. In developing interactive graphics systems that can support large applications, it is surely important to first identify simple and powerful principles for their design. More sophisticated tools *per se* are no panacea. Experience has shown that developing tools based upon ostensibly less powerful techniques can enhance the range of application: c.f. [7]

" ... the use of geometric constructions eliminates the need for solving large systems of non-linear equations inherent in declarative constraint-based systems. Consequently, L.E.G.O. can be used to model comparatively more complex objects."

A framework within which existing approaches to interactive graphics can be interpreted and integrated must depend upon generic techniques with wide - if not universal - applicability.

Unification of principles in interactive graphics appears hard to attain. The methods used are notoriously diverse; there is a conspicuous lack of formalisation and standardisation, and *ad hoc* techniques are prevalent. The problem of formulating an abstract view is compounded by the very different functions that graphical images can serve: as in computer-generated art, medical image analysis, symbolic representations in architecture or circuit design, or technical drawing. The many different types of hardware and software support for graphics pose complementary problems. Nor are existing techniques for formal specification necessarily well-suited for interactive applications.

This paper aims to appraise current progress towards supporting interactive graphics within the framework of a general-purpose programming paradigm based upon definitions: "definitive programming". Previous papers dealing with related work include [1,2,3]. Though many of the issues addressed were considered in [2], there have been several subsequent developments that motivate a re-examination. As described in [4,5], the concept of definitive programming has itself been enriched through the identification of an abstract machine model that can support far more sophisticated computation than the "pure definitive notations" of [1]. This significantly enhances the scope for representing dynamic data relationships, and dealing with concerns - such as animation - where control structures are required. Of additional interest is the parallel development of interactive graphics systems based upon geometric constructions [7,13]: work that can be directly related to the definitive programming approach - with mutual benefit. For instance, definitive programming provides a much broader perspective within which to consider the use of "imperative constraints", whilst research on construction-based modelling suggests new and better solutions to the technical problems of dealing with complex operators raised in [2].

2. Fundamental issues for interactive graphics

The focus in this paper is upon graphics systems that involve interaction in a significant sense. A system that supports "visual programming" is not necessarily such a system: the main function of a graphical interface may be to permit convenient editing of a program that routinely generates graphical images in an autonomous way. To invoke the concept of "significant interaction" is itself to beg a question: "Is it possible to make an objective claim that one system supports richer interaction between the user and the computer than another?" This paper argues that it is - provided that the reader will accept the thesis that a spreadsheet gives better support for interactive calculation than a pocket calculator.

To appreciate the context in which "interaction" is being interpreted, the user may think of a major design project involving the abstract description and evaluation of a complex artefact such as a building. A very large number of drawings may be developed in the design process, and the timescale may be such that drawings have to be revised many months after they are first drawn. Naively, the user will need to be able to specify relationships between objects and components of objects that are to be stored and maintained by the computer. Several key issues arise: How are the relationships to be established? When is the maintenance of relationships carried out? To what extent are relationships maintained through autonomous action on the part of the computer? How is the current status of the design to be represented to the user? How can data relationships be represented in such a way that semantic analysis or simulation can be conveniently performed?

The demands that a user would ideally wish to make upon a design system are very great:

- relationships should be easily perceptible, and conveniently modified
- it should be possible to record partial information about relationships conveniently
- relationships should be expressible at many levels of abstraction
- it should be possible to accommodate a temporary failure to meet constraints, to compensate at a later stage, and conveniently make consequent changes retrospectively.

For the user, a transaction with the system will typically be an incremental change - a single addition to the sequence of perhaps many thousands of previous transactions. The architect who moves a wash-basin a metre nearer to the door will not expect the plans for the entire building to be reprocessed, but might reasonably expect to be advised that the waste pipe is obstructed by a balcony on the floor below. Many existing programming paradigms for interactive graphics are better suited for the complete reappraisal of a design after every trivial amendment than to modest and selective processing proportionate to a minor transaction. To some extent, this is a symptom of a more profound difficulty: how to represent relationships between objects so subtly that a trivial action has some impact, but not too much. Such problems of maintaining relationships that are both consistent and admit a multitude of very minor modifications have promoted the development of systems that avoid the embarrassment of user-intervention wherever possible. Thus a sophisticated design system might invoke a computational scheme of enormous ingenuity that involved reconfiguring all the waste pipes and balconies on an entire wall, at considerable computational cost, and perhaps in violation of the user's aesthetic preferences.

3. Current approaches

The above discussion indicates that the effective representation of geometrical relationships is essential for significant interaction in a graphics system. Many existing approaches fail to address this issue adequately. When the emphasis is on supporting a repertoire of drawing operations (as in say MacDraw), the graphical image is described by the cumulative effect of procedural actions, and the geometric relationships that can be established are limited to the independent manipulation of explicitly defined groups of points and lines. Within such a framework it is not possible to support relationships between geometric entities at an appropriate level of abstraction.

It is clear that an interactive graphics system that gives effective support for abstract relationships must be based upon higher-level primitives. Construction-based modelling recognises this by adopting fundamental geometric constructions as the primitive operations, and describing structural relationships in terms of these. The formulation of data relationships over an appropriate system of data types and operators at a reasonable level of abstraction is common to many approaches: their difference lies in the way that this "underlying algebra" is exploited. Four interrelated approaches to modelling geometric relationships are particularly relevant to this paper:

- functional programming principles (c.f. [9] p255),
- equational / constraint-based principles (c.f. [6]),
- construction-based modelling [7,13],
- definitive programming principles [2,3].

Systems to represent planar line-drawings within these paradigms would typically be based upon an underlying algebra consisting of scalars, **points**, **lines**, and **shapes** comprising multisets of points and lines, together with elementary geometric operators e.g. specifying the line joining a pair of points, the angle between two lines, or the union of two shapes. A brief outline of the different superstructures that can be built upon this foundation within these paradigms will be useful.

Following [9], a functional programmer might represent an abstract **shape** by a function

$$\mathbf{point} \times \mathbf{point} \rightarrow \mathbf{shape}$$

(i.e. as the set of **point**s and **line**s determined by a choosing a particular coordinate system as represented by a pair of **points**), and define complex **shapes** by applying higher-order functions to basic **shape**s. During interaction, the current status of the functional programming system will be determined by what functions have been defined, and the image currently depicted will be determined by what function evaluations have been performed. Geometric relationships are specified and modified by editing a script that defines appropriate higher-order functions, and redisplay is effected through function re-evaluation.

In a constraint-based system, geometric entites are represented by variables, and geometric relationships are expressed as equational constraints between variables over the underlying algebra e.g. insisting that four points lie on a circle, or that two lines are parallel. Ideally, these constraints are considered to be purely declarative in nature: they prescribe the relationships that must hold without burdening the user with details of how they are maintained. Interaction involves editing the current set of constraints, invoking a constraint-satisfaction process that first attempts to reconfigure the geometric entities appropriately, and subsequently leads to their redisplay.

In a construction-based modelling system, the relationships between data are formulated through a sequence of "imperative constraints". In effect, the locations of geometric entities are specified relative to each other through explicit prescriptions that are directly or indirectly expressed in terms of the operators of the underlying algebra. The constraints established in this way are conceived and stored as procedural fragments that encapsulate the description of a graphical image, and can be edited for purposes of reconfiguration or re-display.

In a definitive programming idiom [2], the **point**s, **line**s and **shape**s that make up the graphical image are represented by a system of variables of the appropriate type. The value of a variable can either be specified explicitly, or implicitly by an algebraic expression in terms of the values of other variables and constants. The system of variable definitions is free of cyclic reference, so that the variables can be partially ordered by a data dependency relation:

$v \leq w$ if the value of w is defined - directly or indirectly - in terms of the value of v.

Every part of the image is associated in this way with a variable, and the interrelationship between parts is expressed through the defining formulae. Interaction is effected through editing the current system of definitions, re-evaluation of those variables whose values may be affected, and redisplay of the associated entities.

4. Limitations of current interactive systems

The different programming paradigms for interactive graphics briefly sketched above are often ostensibly evaluated and compared on performance considerations. For instance, construction-based modelling is primarily seen as avoiding the technical problems posed by complex constraint satisfaction in constraint-processing systems (c.f. [7] §3.1). It will be argued in this paper that *performance considerations apart* neither a functional nor an equational programming idiom is an appropriate basis for interactive systems for major applications, and that the merits of construction-based modelling are in part related to a separate concern: the coincidental advantages that result from formulating geometric relationships in a way that *de facto* exploits systems of variable definitions. To dispel any impression that this links definitive programming more closely to construction-based modelling than to the other paradigms, it should be added that definitive programming is in certain respects more akin to functional and equational programming; it merely introduces one significant abstract concept (viz variable definition) that can be very directly linked to the use of geometric constructions.

The evaluation of interactive systems proposed here is based upon the criteria for significant interaction outlined in §2 above. The questions that are most relevant to the user's concern in interacting with a graphics system may be formulated as: "What are the current relationships between entities? How can I reference entities? How can I modify the relationships?" The architect who moves the wash-basin may not appreciate the implications of this simple action. In a well-conceived system, it is to be expected that some effects of actions consequent upon moving the basin - such as moving an associated mirror - are performed automatically, but such automatic responses have their limitations, and may need to be activated at the user's discretion. In a badly conceived system, moving the basin might involve changing the dimensions of the building to accommodate a dislocated balcony at the corner of an external wall. Such an illustration highlights the very significant part played by data dependencies in an interactive system: it may be exceeedingly hard to anticipate what entities can be referenced in isolation, and what the implications of modification might be.

How can issues of reference of this nature be handled in existing paradigms? Irrespective of which paradigm is used, it seems clear that a solution must involve making the program code that generates an image transparent to the user. For instance, to give informal support to the architect who wishes to understand the consequences of moving the basin, it would be necessary to highlight those parts of the code pertaining to the basin, to indicate in some way how modifying this code would affect the rest of the program, and - to ensure a truly interactive rather than batch processing environment - equip the compiler to interpret a local change to the code without complete recompilation. The effectiveness of a programming paradigm for interaction may be better gauged by how easily these processes can be performed than by traditional criteria. Paradoxically, the fundamental principles that are represented as special virtues of good programming idioms, viz referential transparency in declarative systems, and modularity and information hiding in sophisticated procedural systems, do not assist the recognition of implicit data dependencies.

The notion of "moving the basin" is itself rooted in a procedural computational framework. Conceptually, there is a specific variable **wash_basin** whose value is an image of a basin on the architectural plan, and whose value is changed when the basin is moved. In the L.E.G.O. idiom, the basin might itself be represented by a procedure including a parameter specifying its position. By editing this procedure definition, it would certainly be possible to relocate the basin in isolation, but to model the relationship between the basin and the mirror would require some data dependency between the procedures drawing the mirror and the basin. From an implementation perspective, an object-oriented paradigm may clarify this model, making it possible to represent the basin by an object rather than a family of primitive procedural variables, and the geometric relationship between the basin and the mirror through a message passing protocol, but the data dependencies established in this way will nonetheless be difficult to ascertain.

Within a purely declarative programming paradigm, there is - ideally - no direct way to model procedural variables such as **wash_basin**: a program is a set of functions, or a system of equations, that defines a set of values through evaluation, or solution. In this respect, the philosophical foundations of declarative paradigms militate against rich support for reference. To modify the graphical image the user must edit the set of functions or equations, so that in principle a totally new program has to be interpreted. Though some special provision for modularisation could be made by elaborating the specification (c.f. [10]), these are unlikely to enable the user to view the specification flexibly as incorporating references to parts of the image that are subject to change.

In a constraint-based framework, there are variables to represent geometric entities, though it may not be easy to choose these at an appropriate level of abstraction. Even if the problem of identifying the constraints that pertain to the wash-basin can be solved, formidable difficulties remain. Determining whether the simplest new constraint leads to inconsistency is presumably algorithmically undecidable in general, so that its impact can neither be conveniently represented to the user, nor acknowledged by the compiler.

Within the definitive programming paradigm, it is still necessary to determine data dependencies through examination of the program code. The difference is that the use of definitions makes these dependencies transparent, in that they can be precisely identified syntactically, and readily interpreted. These virtues are epitomised in numeric rather than geometric appplications by the spreadsheet, which allows relationships and values to be defined and modified in an easily comprehensible manner. This has important ramifications, to be investigated below.

5. Geometric constructions and definitions

To explore the significance of using definitions to formulate geometric relationships, it will be helpful to recast a construction-based specification in definitive terms. This is easily done for the simple example of a geometric construction "bisection of a line", as formulated in L.E.G.O in [7]:

```
(point 400 370 A)                 point a = {400,370}
(point 600 470 B)                 point b = {600,470}
(line A B L)                      line l = [a,b]
(circle A L C1)                   circle c1 = circle of radius l with centre a
(circle B L C2)                   circle c2 = circle of radius l with centre b
(intersection C1 C2 X1 X2)        pointset {x1,x2} = intersection (c1,c2)
(line X1 X2 P)                    line p = [x1,x2]
```

The left-hand column is the original L.E.G.O specification; the right-hand an equivalent formulation using a fictional variant of the definitive notation DoNaLD [2,3]. Note that the "operators of the underlying algebra" required for this definitive notation correspond precisely to the primitive constructions of L.E.G.O. The strong syntactic resemblance between the two specifications should not detract from very fundamental semantic distinctions. The order of the definitions on the right is not significant: the data relationships that they express is intrinsic in the variable references. The definitions are not to be interpreted as a sequence of constructions to be carried out, but as an explicit description of the data dependencies between geometric entities. Such systems of definitions can form a component of a larger definitive specification irrespective of whether the variables to which they refer are currently well-defined. The effect of redefining a variable is to update that part of the specification that is linked through data dependency to the variable, and to lead - through re-evaluation - to selective redisplay.

The above illustration may suggest that the distinction between the L.E.G.O. and DoNaLD approaches is cosmetic. After all, the difference between the two code fragments above is small enough for easy re-interpretation in either direction. It can be argued - for instance - that storing a set of definitions that describes the bisector of the line AB when A and B denote the same point, is essentially the same as storing a procedural file of instructions that happens at present not to be executable. The data dependencies between variables in the L.E.G.O. specification are in this case easy to identify, and incremental recompilation of the procedural specification after any single statement were to be modified (as in "redefining a variable") is well within the scope of current compiler technology [14].

The virtues of the new perspective afforded by definitive programming cannot be fully appreciated from such a small example. To justify the L.E.G.O. specification as a system of "imperative constraints" requires an interpretation of the variables different from that used in a traditional procedural approach, lest they exist only whilst the drawing procedure is being executed. The reinterpretation of construction steps as definitions captures this distinction, and exposes the characteristics of a small-scale L.E.G.O. specification that make it conceptually simple for the *user* to interpret. As the history of procedural programming testifies, a procedural fragment can become very hard to interpret in the context of a large specification. On these grounds, a definitive reformulation seems to be a promising way to express the particular qualities of specifications in the L.E.G.O. idiom, provided that it can be shown that definitive programming in principle has the expressive power to support major applications.

6. Towards definitive programming in the large: data representation

How do definitive programming principles scale up? There are two aspects to be considered: the representation of data, and the specification of control. The issue of treating complex data types within a definitive framework has received much attention in previous papers [1,2]. The principle is clear: defining the value of a variable of a complex data type should be possible at many different levels of abstraction, so that *either* a recipe for the entire value of a variable is supplied *or* the variable is composed of a family of independently defined variables whose values describe its constituent parts. Two methods of dealing with this issue have been developed: the use of "moding" as in the definitive notation ARCA, and the use of **openshape** variables that permit a hierarchical definition of sets of **point**s and **line**s as in DoNaLD [3,4]. The principal outstanding problem is describing a formal framework in which to treat the specification of operators of a complex type: a topic that provides a natural context for the further consideration and comparison of functional, construction-based and definitive methods.

The issues are well illustrated by considering the design of user-defined operators of type **shape** within DoNaLD. The naive view is that a user-defined operator of type **shape** is an extension to the underlying algebra, and should be described by a pure function returning a value of type **shape**. The idea of using a definitive notation to specify the function itself is superficially unattractive - within the function body, there seems to be no purpose in having variables whose values are specified by definitions. In the original DoNaLD design [3], this was the position adopted: a new **shape** operator should be a function without side-effects to be specified using a simple procedural or functional notation.

Such a convention is not wholly satisfactory. A typical use of the **shape** operator f(a,b,c,...) would involve a **shape** variable declaration followed by a definition:

shape S; S=f(A,B,C,...).

The problem then arises: in making use of the implicitly defined variable S, what references to the constituents of S are valid? For instance, if f returns a set of points and lines that defines a square, how is it possible to reference its edges and vertices? Oddly enough, though functional programming might appear to be the obvious paradigm to choose for the specification of a pure function, it is unhelpful in this respect. It seems that the problems associated with interactive revision of a functional specification referred to in connection with interaction in §4 above also impinge where only *reference* to a specification rather than revision is involved.

A trivial illustration will clarify the issues, and indicate why the present proposal for the specification of DoNaLD **shape** operators is as much influenced by the ostensibly procedural treatment of function definition in L.E.G.O. as by functional programming ideas. Following [9], a system of lines resembling the rungs of a ladder might have the following functional specification:

ladder: N×V×V → P

ladder(n,a,b) ≡ **if** n=1 **then** line(a,b) **else** picture(line(n.a,b),ladder(n-1,a,b)).

In this specification, a and b are parameters representing vectors, n is an integer to specify the number of rungs, and P designates the data type "picture". It is an elegant and concise description, but is many ways inappropriate in an interactive graphics setting. To depict the rungs of a ladder, a function evaluation such as

ladder(n,<1,2>,<2,1>)

must be performed. To reference a particular rung of the ladder, as might be required in simulating "climbing the ladder", is impossible without embellishment of the original specification. To model a real ladder, that might consist of two congruent hinged sections that could be locked into a V-shaped or linear relation, it would not be enough to use such a functional specification - even in conjunction with a DoNaLD **shape** variable. Of course, it is not infeasible to elaborate functional methods to partially redeem the situation (c.f. [10]), but there appear to be fundamental limitations associated with the strictures of referential transparency.

By contrast, a procedural approach to the specification of the ladder() function makes the problems of attaching references to the individual rungs of the ladder less acute. In effect, it is relatively to easy to formulate a procedural description of a function that returns not only a value of type **shape** to represent the ladder, but a family of variables of type **line** to represent the rungs. To illustrate the form that such a specification might take, consider the two following specifications - the one written in the construction-based L.E.G.O. idiom, the other in the definitive DoNaLD style:

```
(define_function ladder(n a b))
(line  n.a  n.a+(b-a)  l)
(write_function)
(if (n>1) then
        (ladder n-1 a b)
(end_function)
```

```
openshape ladder(int n, point a,b)
within ladder {
        line L = [n.a, n.a+(b-a)]
        if n>1 then {
                shape Ladder = ladder(n-1,a,b)
        }
}
```

The same considerations that applied to the comparison of the "bisection of the line" construction discussed above again apply. The dual semantics that makes it possible to view a L.E.G.O. specification from a procedural and a definitive perspective is helpful here in interpreting the specification on the right as defining an operator together with a system of references to the value it returns. By introducing such operator specifications it is possible to formulate a DoNaLD definition:

shape Ladder; Ladder = ladder(7, {1,2},{2,1})

so that Ladder/L refers to the top rung, Ladder/Ladder to the remaining set of rungs, of which Ladder/Ladder/L is the topmost etc. This is the kind of referencing facility that is required when establishing relationships between complex objects that cannot conveniently be explicitly defined.

6. Definitive programming in the large: control

In this paper, the case for "programming with definitions" has so far been made on technical grounds. The primary argument has been that conventional interactive graphics involves editing functional, procedural or constraint-based specifications that typically correspond only in a very obscure way to the cognitive models the user requires. Particular attention has been focussed on the limited way in which conventional paradigms for interactive graphics support references to objects and their constituent parts. Some of the problems can be attributed to the vague semantics of variables, and the dichotomy between declarative frameworks in which references are too inflexibly established, and procedural environments in which they are too impermanent. The result is that interactive systems are frequently ill-equipped for significant user intervention.

In what respects does definitive programming potentially offer better prospects? To an extent, the ideas introduced above meet the need for better support for the abstract description and referencing of graphical images, and allow incremental changes to relationships to be subtly and efficiently represented. The representation of dependencies between data using definitions is a particularly significant concept, since it enables the user to prescribe, and to anticipate, the effects of amending a specification.

The significance of using definitions to support interaction goes beyond technical considerations alone, however. A central thesis of current work on definitive programming is that a definitive system is a useful cognitive model not merely for user-computer interaction, but for the actions of an agent in a concurrent system [4,5]. In effect, a single system of definitions is best conceived as articulating the effects of a particular action or intended action. Only by such a generalisation of the concept of "definitive notations for interaction" does it become possible to express the fact that an architect's expectations on moving the wash-basin might or might not encompass an automatic compensating movement of the mirror, and could lead to the invocation of an automatic process to reconfigure the external balconies.

As this illustration suggests, though it may not be possible for the user to predict the outcome of an automatic computation intiated during an interaction, it is necessary to give the user both a degree of control over such computation, and a good appreciation of its effects. The perspective taken in this paper is that the description and manipulation of geometrical relationships throughout the entire user-computer interaction should be seen as one consistent (non-terminating) computation comprising many simple transactions. In this interactive process, the user and the computer participate via sequences of homogeneous actions, and the effect of each individual action is transparent to the user. This means, in particular, that the user knows the effective geometric relationships throughout the entire interaction, even to the extent that it is possible for the user to suspend and intervene during an automatic computation.

The appropriate abstract machine model adopted for this purpose is the abstract definitive machine (ADM), as described in [4,3,5]. A full discussion of the ADM is beyond the scope of this paper, but a brief sketch of its use in the animation of a simple concurrent system will be used to illustrate the key ideas. In many respects, this ADM simulation resembles examples developed using the NoPumpG software [12].

Suppose that two blocks b1 and b2 are connected by a string of length d, and that the blocks are independently controlled by two handlers. The behaviour of the pair of blocks can be described by the ADM program in Figure 1. The handlers are represented by entities: each entity comprises a set of variable declarations and definitions, and a set of guarded actions. In this context, each action consists of a sequence of redefinitions. The operation of the ADM is such that in each machine cycle the guards of all actions within currently instantiated entities are evaluated, and the actions associated with true guards are performed in parallel (subject to non-interference). The entity handler() for instance, may at any stage be holding b1 - as defined by the variable h1, and when holding the block may be pushing left (pl1) or right (pr1). The movement of the blocks is determined by the bmover() entity, and depends upon the directions in which the blocks are currently being driven (dr1, dl1, dr2, dl2).

The specification of bmover() illustrates how the context within which variables are redefined must be altered to reflect conditions such as whether the blocks are touching (t12), and whether the string is taut (st). For instance, if b1 is being driven to the left , and the string is taut, the action "dl1 and st -> p2=p1+d; p1=|p1|-1"

is enabled, indicating that the movement of b1 to the left must necessarily drag b2 to the left also. As it stands, the specification does not include any method for resolving interference between actions, as for instance would arise if b2 were simultaneously being driven to the right (c.f. [5]).

The entity bstate() records the current position and status of the blocks: their positions (p1, p2) and whether the string either should snap or has been snapped (nostr).

The simulation is very simply animated by complementing the ADM specification with a file of DoNaLD definitions to describe the required display. The DoNaLD definitions of the **openshape**s b1 and b2 that represent the blocks, and the **line** str that represents the string can be abstractly viewed as an additional entity within the ADM specification. The block display registers the location of the blocks (as specified by the variables p1 and p2), and the handler-block relationship (as specified by h1, dl1, dr1 etc). The simulation snapshot depicts a situation in which b1 is being driven to the right, and the block b2 is being held, for instance.

Concluding remarks

This paper has argued the case for giving serious consideration to the development of definitive programming as a medium for describing interactive systems, and perhaps indicated something of its potential. There are many issues still to be explored, and work is in progress on a variety of related topics, including applications to CAD, and to the simulation of concurrent systems.

Good semantic models for graphics and interaction deserve closer consideration. There are many indications in this paper of the apparent inadequacy of our present theoretical foundations for programming in respect of data representation and manipulation (c.f. [11]). An intriguing historical sidelight on this issue is the demise of the concept of a mathematical variable as "representing a variable quantity" during the arithmetisation of analysis in the 19th century [8]. Perhaps the development of better methods of programming for interactive graphics can stimulate reconsideration of the formal status of the rich - if obscure - concept of *variable* that geometrical intuition formerly inspired.

Acknowledgements

I am indebted to Mike Slade and to Edward Yung for their support in designing and implementing software prototypes for DoNaLD and the ADM, and to Steve Russ for many helpful discussions on the subject of reference and interaction.

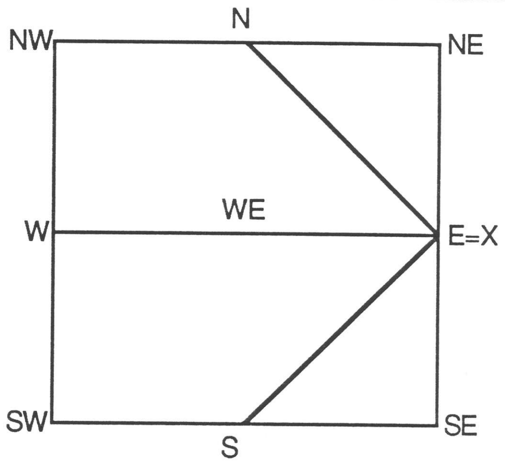
References

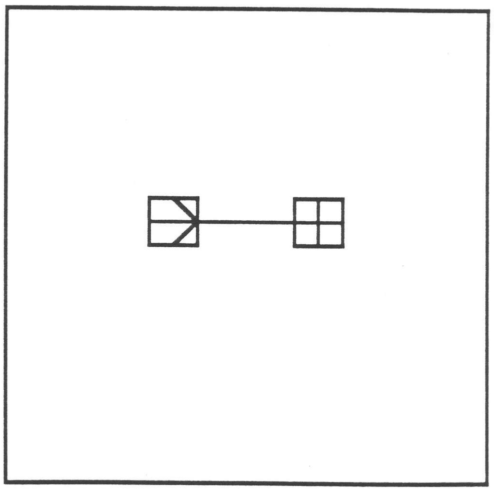
1. W M Beynon *Definitive notations for interaction*, Proc hci'85 CUP 1985, 23-34
2. W M Beynon *Definitive principles for interactive graphics*, NATO ASI series F:40, 1987, 1083-1097
3. W M Beynon, D Angier, T Bissell, S Hunt *DoNaLD: a line drawing system based on definitive principles*, Univ of Warwick RR#86, 1986
3. W M Beynon, Y W Yung, *Implementing a definitive notation for interactive graphics*, New Trends in Computer Graphics, ed N Magnenat-Thalman, D Thalman Springer-Verlag 1988, 456-68
4. W M Beynon, M D Slade, Y W Yung *Parallel computation in definitive models*, in Proc Conpar'88 (to appear)
5. W M Beynon *Definitive programming for parallelism*, CS RR#132, Warwick Univ, 1988
6. A Borning *The programming language aspects of ThingLab, a constraint-oriented simulation laboratory*, ACM Transactions on Programming Languages 3(4), 1981, 353-387
7. N Fuller, P Prusinkiewicz *Geometric Modelling with Euclidean Constructions*, New Trends in Computer Graphics ed N Magnenat-Thalmann, D Thalmann, Springer-Verlag, 379-391
8. P Geach, M Black *Philosophical Writings of Gottlob Frege.*
9. P Henderson *Functional Programming*, Prentice-Hall International 1980
10. J Hughes *Why Functional Programming Matters*, PMG Report #16, Chalmers Univ of Tech & Univ of Goteborg, 1984
11. W Kent *Data and Reality*, North-Holland 1978
12. C Lewis *Using the NoPumpG primitive*, Dept of Computer Science and Inst of Cog Sci, Univ of Boulder
13. T Noma, T L Kunii, N Kin, Henomoto, E Aso, T Yamamoto *Drawing Input Through Geometrical Constructions: Specification adn Applications*, New Trends in Computer Graphics, ed N Magnenat-Thalman, D Thalman Springer-Verlag 1988, 403-415
14. T W Reps *Generating Language-Based Environments* MIT Press 1984

Figure 1:
A block moving simulation illustrating definitive principles

```
entity  handler1()
{
     definition
         d1 = dl1 or dr1,
         dl1 = h1 and pl1, dr1 = h1 and pr1,
          pl1 = 0, pr1 = 0, h1 =0
    action
         not h1 -> h1=1,
          h1 and not d1 -> h1=0: pl1 =1: pr1=1,
         dl1 -> pl1 = 0, dr1 -> pr1 = 0
}

entity  bstate()
{
      definition
         p1, p2, d,
         st = not nostr and (p2-p1)==d,
         t12 = (p2-p1)==1,
         nostr = 0
     action
          not nostr and (p2-p1)>d -> nostr=1
}

entity  bmover()
{
       action
           dl1 and not st -> p1=|p1|-1,
           dl1 and st -> p2=p1+d; p1=|p1|-1,
           dr2 and not st -> p2=|p2|+1,
           dr2 and st -> p1=p2-d; p2=|p2|+1,
           dr1 and not t12 -> p1=|p1|+1,
           dr1 and t12 -> p2=p1+1; p1=|p1|+1,
           dl2 and not t12 -> p2=|p2|-1,
           dl2 and t12 -> p1=p2-1; p2=|p2|-1,
}

bstate(); bmover(), handler1(); handler2()
```

```
openshape b1
within b1 {
    point O
       O = {500+~/p1*100, 500}
    point NE,NW, SW,SE
      NE = O + {50,50}
      NW = O + {50,-50}
    ...........
    line  n,s,e,w
      n = [NW,NE]
      s = [SW,SE]
    ...........
    point N, E, S, W, X
      N = if ~/h1 then (NE+NW) div 2 else O
      S = if ~/h1 then (SE+SW) div 2 else O
      E = if ~/h1 then (NE+SE) div 2 else O
      W = if ~/h1 then (NW+SW) div 2 else O
      X = if ~/dr1 then E else
                    if ~/dl1 then W else O
      line WE, NX, SX = [W,E], [N,X], [S,X]
}
int  p1,h1,dr1,dl1,p2,h2,dr2,dl2,nostr
line  str
str = [b1/E, if nostr then b1/E else b2/W]
```





Above: The ADM control program

Bottom right: Simulation snapshot

Middle right: Detail of the block1 display

Top right: DoNaLD display specification