Implementing a definitive notation for interactive graphics

Meurig Beynon and Edward Yung,
Dept of Computer Science,
University of Warwick,
Coventry CV4 7AL

ABSTRACT

This paper describes the application of a definitive (definition-based) programming paradigm to graphics software. The potential merits of using definitive principles for interactive graphics were considered from a theoretical perspective in [Be87]; this paper is complementary, in that it describes the insights gained through practical experience in implementing a prototype system. The main characteristics of the prototype implementation are illustrated by simple examples. Analysis of the abstract machine model underlying this implementation suggests a general purpose programming paradigm based on definitive principles that can be applied to more ambitious applications.

Introduction

This paper describes the application of a novel programming paradigm to graphics software. The programming paradigm - "definitive programming" - is based upon the use of definitions for interaction. The potential merits of using definitive principles for interactive graphics were considered from a theoretical perspective in [Be87]; this paper is complementary, in that it describes the insights gained through practical experience in implementing a prototype system.

The notion of using definitive notations for interaction was first described in [Be85]. The essential principle in developing such a notation is to devise an "underlying algebra" of data types and operators which reflects the universe of discourse, and to introduce appropriate variables to represent values in the underlying algebra, either explicitly or implicitly, through a defining formula. In such a system, a **pure** definitive notation, a program - or **dialogue** - essentially consists of a sequence of variable definitions and evaluations. Perhaps the simplest example of such a notation is obtained by choosing the underlying algebra to be traditional arithmetic, when the dialogue resembles the use of a spreadsheet, stripped of its tabular interface.

As explained in [Be85], definitive notations appear to be very well-suited for dialogue. They make it possible to represent the state of a dialogue effectively, since the combination of implicit and explicit definitions allows both persistent relationships, and transient values, to be recorded. An important feature of a dialogue over a definitive notation is that all the information needed to determine the current state of the dialogue is automatically stored, and can be recovered by interrogating the variables to obtain their current definitions.

It is not our purpose in this paper to describe the detailed design of programming notations based on definitive principles; the illustrative examples below make use of abstract notations adapted from DoNaLD ("a definitive notation for line drawing"), and EDEN ("an evaluator for definitive notations"), and the interested reader is referred to [Be86a] and [Yu87] for more details. Our emphasis is upon introducing general principles and techniques that can be used to implement definitive notations for graphics, and - pursuing the directions suggested in [Be87] - can be developed to support much more sophisticated systems and applications than are illustrated here.

The paper is in two main sections. §1 introduces the principal features of the graphics notation DoNaLD, and illustrates how it may be used to describe a room layout so as to allow interactive experimentation with furniture configurations and room dimensions. §2 introduces the programming language EDEN, intended for the implementation of definitive notations. It also

includes some program fragments showing how a DoNaLD dialogue is translated into EDEN. An examination of the abstract machine model underlying EDEN suggests a general purpose programming paradigm based on definitive principles. Some directions for future research directions are also outlined.

§1 DoNaLD: a definitive notations for line drawings

1.1 Basic principles

The DoNaLD notation (a "Definitive Notation for Line Drawings") is intended for the interactive display and manipulation of planar diagrams comprising points and lines. As a definitive notation, it is based upon an underlying algebra comprising five data types: **integer**s, **real**s, **point**s, **line**s and **shape**s, and a variety of simple geometric operators. Scalar values are represented by **integer** or **real** variables, points in the plane by **point** variables, directed line segments (that is, lines defined by an appropriate pair of endpoints) by **line** variables, and line drawings comprising a multiset of **point**s and **line**s, together with a set of **real** and **integer** attributes by **shape** variables. Following the usual pattern, a DoNaLD dialogue then consists of a sequence of declarations of variables, definitions of variables of the form:

variable = *formula*

specifications of user-defined operators, and evaluations of variables. For this purpose, realising the line drawing represented by a **shape** variable is viewed as a special kind of evaluation.

The full details of the operators in the underlying algebra appear in [Be86a]. In brief, there are standard arithmetic operators, vector operators acting upon **point**s viewed as 2-dimensional vectors, and a variety of operators on points and lines. The latter include constructors to synthesise a point in the plane from its component coordinates, and a line segment from its endpoints, and selectors to extract coordinates from **point**s and endpoints from **line**s in the usual fashion. There are also geometric operators for rotating and scaling **shape**s, and an operator for combining two or more **shape**s.

1.2 Dealing with complex data types

The introduction of complex data types into the underlying algebra poses some problems for the definition and reference of variables (cf [Be85] and [Be87]). The solution to the reference problem for **shape**s adopted in DoNaLD is based upon a representation of a line drawing by a **shape** variable resembling that of a file system directory as a union of files and subdirectories. Such a representation corresponds to an abstract view of a line drawing as a union of points, lines and sub-drawings. In effect, it is based upon a recursive specification of the **shape** data type, viz:

shape = set of **real** / **integer** attributes + set of **point**s + set of **line**s + set of **shape**s.

There are two kinds of variable of type **shape**; these are declared as "**shape**" and "**openshape**" variables, and are broadly analogous to "abstract" and "explicit" variables as described in [Be85]. The value of a **shape** variable is to be defined implicitly by means of an expression of type **shape**. An **openshape** variable, which resembles a directory, is composed of constituent **real**, **integer**, **point**, **line**, **shape** and **openshape** variables, and its value is defined componentwise by associating values with its constituent scalar attributes, points, lines and subshapes.

Each variable is either declared globally, or denotes a constituent of an **openshape** variable, which may itself be a variable of type **shape**. The value of a **shape** or **openshape** variable which is declared as a constituent of an **openshape** variable X is a *subshape* of the value of X, comprising a subset of the set of **point**s and **line**s associated with X. The authentic variable name is that used to reference the variable from the global context, and is in general specified by a sequence of openshape variable names separated by '/'s to identify the enclosing openshape, followed by a local name to identify the appropriate constituent. (The syntax resembles filenames in UNIX directories.) A variable declaration thus takes the form

type var_name

where *type* is **integer, real, point, line, shape** or **openshape**, and *var_name* is of the form

context loc_var_name

where *context* is a concatenation

(*loc_var_name* '/')*

in which each *loc_var_name* references an **openshape** variable.

The semantics of **integer, real, point** and **line** variables is straightforward. A declaration of the form

openshape S

identifies S as an *explicit* **shape** variable, which is not itself an *l*-value, but enables the subsequent declaration of attributes and components of S. The subshape S can then be defined componentwise according to the normal semantic rules. In contrast, a declaration of the form

shape V

identifies V as a *virtual* **shape** variable, whose value and component structure must be defined by means of a **shape** expression, and, in particular, cannot be defined componentwise.

1.3 Changing contexts

The syntax of a DoNaLD dialogue is simplified by introducing an analogue of "changing the directory" in a file system; the hierarchical view of a line drawing as incorporating sub-drawings can then be reflected in the manner in which it is defined and referenced.

Formally, the set of variables associated with the **openshape** variable V consists of the points and lines of V, together with local scalar variables and all variables associated with subshapes of V. Definitions of the constituents of **openshape** variables are governed by a single *scope rule*: each subshape V/S of V, and all variables associated with V/S must be defined in terms of variables associated with V. Conceptually, the actions in a DoNaLD dialogue can be viewed as taking place in the context of a single universal **openshape** variable U, whose constituents are the **point, line, shape** and **openshape** variables not contained in any user specified **openshape** variable.

The construct used to specialise the dialogue to a particular context, and provide for all references to variables to be interpreted relative to an **openshape** other than U is:

within *context* { ..... }

where a general sequence of actions, possibly including further **within**-clauses, is specified between the braces.

The scope rules permit a variable v within the openshape S (and necessarily not within any **openshape** subshape of S) to be defined in terms of variables defined in the enclosing context for S, rather than within S itself. To avoid having to leave the context of S in order to make such a definition, each variable associated with the enclosing context for S in the expression defining v can be prefixed by an escape symbol "\", to indicate that it is to be interpreted with reference to the enclosing context for S.

1.4 The user interface

A typical DoNaLD dialogue includes many definitions, not all of which can be conveniently displayed at once, reflecting different levels of abstraction in the description of a picture. The user-interface for DoNaLD is designed to reflect the way in which relationships at each level of abstraction are captured by a set of definitions, and is based upon a family of windows associated with **openshape** variables.

With each **openshape** variable, there is an associated context window, in which the appropriate local definitions are displayed together with the names of any local **openshape**

variables, and through which the relationships within that context can be viewed and manipulated. The context windows are organised in a tree structure reflecting the hierarchical relationships between contexts. At the highest level of abstraction, and the root of the tree, there is a global window, in which all the declarations and definitions of variables at the outermost level are displayed. Each **openshape** variable in the global context defines a subcontext at the next level of abstraction, and its associated window is a child of the global window.

At any given stage in a DoNaLD dialogue, a number of windows can be open, but only one window is currently active. In general this is the window associated with the current context. In effect, all declarations and definitions made whilst a particular context window is active are interpreted as being in the scope of a **within** *current_context* clause. Each context window has a header to specify the **openshape** variable name relative to the universal context, and a footer comprising a set of "buttons" which are used for interrogation of variables or to change the current context, together with a dialogue box in which new declarations and definitions can be entered. Figure 2 illustrates how the definitions in Figure 1 might be presented via such an interface (for more details, see [Be86a].)

1.5 An illustrative example

The form of the basic notation introduced above is illustrated in the abbreviated dialogue depicted in Figure 1 below. A dialect of DoNaLD has been used; this makes use of a simple subset of the DoNaLD operators, but incorporates additional features for handling simple geometrical constraints. Figure 2 gives full details of the definitions in the context window associated with the **openshape** *room*, and the corresponding graphical display. (The dotted lines are used to indicate the extent of the geometric objects defined by the variables *door* and *desk*.) Figure 3 depicts the context windows of the **openshapes** that are introduced in the dialogue of Figure 1.

In Figure 1, the global context for the dialogue is the **openshape** *room*. The room is defined to be rectangular. The definitions used for this purpose include implicit definitions of the wall represented by the **lines** *E*, *S*, and *W* in terms of the corners of the room (the **points** *NW*, *NE*, *SW* and *SE*), and of the four corners in terms of the explicitly defined width, length and centre of the room. The remaining wall, represented by the pair of lines *N1* and *N2*, is implicitly defined in terms of the corners *NW* and *NE* and the location and dimensions of the door.

Figure 1 also illustrates the use of simple constraints. In DoNaLD, three types of constraint handling are envisaged; these are syntactically expressed in the form:

impose *B*, **monitor** *B*, **maintain** *B*

where *B* is a boolean variable defined to represent a condition expressed in terms of primitive arithmetic and geometric relationships. Each of the above constructs is to be read as the declaration of a particular type of boolean variable: in effect, "**impose** *B*" is an abbreviation form of:

imposed constraint *B*

Semantically, **impose** *B* ensures that the state of dialogue is never such that *B* is violated; a dialogue action that would lead to such a state is automatically revoked. The use of **monitor** *B* on the other hand permits violation of the constraint *B*, but monitors such a violation - for example, by displaying a warning message in a special monitoring window whilst *B* is violated in the current dialogue state. The use of **maintain** *B* is intended to indicate that a violation of the constraint C through a dialogue action on the part of the user will provoke a computer response that restores a state of dialogue in which C is once more valid. (Maintenance of a constraint in general requires details of the actions prescribing the computer's response to a violation - see [Bo86] and §2.2 below.)

In Figure 1, the declaration of *desk* as an **openshape** within *room* is subject to the imposed constraint that the extent of the variable *desk* is contained in the extent of the variable *room*. (Details of the auxiliary geometrical primitives required to support **subshapes** - such as the extent of an **openshape**, and the boolean operators for **shape** inclusion - are left to the reader's imagination.)

§2 EDEN: an evaluator for definitive notations

2.1 Background to the implementation

The system of definitions and constraints in Figure 1 describes the semantic content of the user's interaction in a direct and simple fashion. The virtue of using a medium such as DoNaLD is that "the semantics of the current dialogue state" has an unambiguous interpretation, and the user-computer interaction can be conceived as a sequence of transitions from one system of definitions to the next. In effect, the procedural elements of the dialogue are encapsulated in dialogue actions, such as updating the parameter *door/open*, at a high level of abstraction. (Compare the representations of current state of the interface described in [Fo87b].)

In implementing such a dialogue, the mechanisms used to handle the textual and graphical displays also have to be considered. Just as in a spreadsheet the displayed values of variables must be continously updated, the graphical display must be kept consistent with the current dialogue state. Other aspects of the user-interface must also be implemented, such as the management of the context windows outlined in §1.4.

It will be convenient to distinguish between those aspects of the interaction that are concerned with the semantics of the application (eg does the door currently hit the desk?), and the more ephemeral issues concerning the current state of the display interface (eg is the context window for the **openshape** *desk* currently open?). In general, many different kinds of interaction must be supported:

user actions that affect the application semantics
(eg *door/open* = true)
computer responses that affect the application semantics
(eg revoking a user action that causes violation of an imposed constraint)
computer responses that are linked to the application semantics
(eg updating the current position of the door on the graphical display)
computer responses that are independent of the application semantics
(eg opening a context window when the user enters the context of another **openshape**)

In our present prototype implementation, the distinction between interaction involving the application semantics and interaction concerned with interface management is reflected in the mode of implementation. For modelling the semantics of the interaction in the application, we translate the DoNaLD dialogue into a lower level definitive notation that forms part of the implementation language. This means in particular that the current state of the semantic dialogue is explicitly modelled (albeit in a modified form) in the implementation. For dealing with the dynamic behaviour of the display interface, we then use conventional procedural programming techniques, linked - where appropriate - to the current state of the semantic dialogue.

In §2.2 below, we examine the advantages of our implementation strategy from a pragmatic perspective. In particular, we outline the main features of the programming language EDEN that has been developed as an evaluator for definitive notations, and illustrate how the dialogue in Figure 1 can be translated into EDEN. In §2.3, we consider the abstract machine model underlying EDEN, and propose a simplification that represents a new general purpose programming paradigm based on a direct generalisation of pure definitive notation. Some of the ramifications are briefly examined in §2.4. A key idea is that definitive principles can be used to describe all aspects of the interaction, not only the semantics of interaction in the application.

2.2 The EDEN language

The EDEN language (an "Evaluator for DEfinitive Notations") is intended as a general purpose software tool to assist the implementation of definitive notations [Yu87]. EDEN includes

the traditional features of a procedural language: in this case, a subset of C. It also supports a definitive notation over an underlying algebra comprising **list**s (as in Lisp) whose atoms are either scalars or strings. The dependencies between the variables in this definitive notation are monitored at all times, and the current values of variables are selectively updated as required. Within EDEN, the user can define **list** functions to augment the underlying algebra, so that form of the **list** definitions can be very general. For convenience, variable type checking is handled automatically in EDEN, and no declarations of variables are required.

EDEN has an additional feature whereby procedural actions, whether in the form of redefinitions of explicitly defined **list** variables, or more traditional invocations of C-like procedures, can be linked to the semantics of the "internal dialogue" in the definitive notation. To this end, the user can specify a procedural action to be triggered when the value of a variable in the internal dialogue is altered, whether directly or as a result of the redefinition of another variable. (For a fuller account of the semantics of EDEN see §2.3 below.)

The above discussion can be illustrated by some simple fragments indicating how the DoNaLD dialogue in Figure 1 is translated into EDEN:

DoNaLD input	EDEN translation
int *width, length* **point** *centre* **point** *NW, NE, SW, SE* **line** *N, S, E, W*	_ **is** [OPENSHAPE, _width, _length, _centre, _NW, _NE, _SW, _SE, _N, _S, _E, _W]; new_object(_centre, _NW, . . .); **proc** P_centre: _centre { plot_point(_centre); } **proc** P_NW: _NW { plot_point(_NW); } . . .
width = 800 *length* = 800 *centre* = {500, 500} .	_width **is** 800; _length **is** 800; _centre **is** [POINT, 500, 500];
N1 = [*NW, door/hinge*] *N2* = [*door/hinge* + {*door/size*, 0}, NE]	_N1 is [LINE, _NW, _door_hinge] _N2 is [LINE, vector_add(_door_hinge, [POINT, _door_size, 0]), NE];
. .	. .
openshape *desk* **monitor** *door_hits_desk* *door_hits_desk* = **intersects**(*desk,door*)	_room_desk **is** [OPENSHAPE, _desk_width, ...]; **proc** monitor_door_hits_desk: _door, _desk { **if** intersects(_door, _desk) **then** print("door hits desk") }
within *desk* { **int** *width* . . . }	_desk_width **is** 200;

Note that in translating DoNaLD definitions to EDEN definitions, the translator prefixes object names with underscores (the identifier associated with the root shape *room* is suppressed). The EDEN definition "_ **is** [OPENSHAPE, . . .]" is equivalent to the sequence of variable declarations in the DoNaLD input; it defines the value of the variable "_" as a **list** whose first element is the type code OPENSHAPE, and whose subsequent elements are implicitly specified by the variables _width, _length, _centre etc whose definitions are introduced at a later stage.

The procedure *newobject* initializes the graphics segments. The syntax

proc *procedure_name* : *trigger* { *procedure_body* }

is used in EDEN to designate a procedure to be executed when the value of a trigger variable is altered. The triggered actions *P_centre*, *P_NW*, . . . are generated by the translator, and serve to update the graphics segments when the values of the DoNaLD variables *centre*, *NW*, ... are altered. (The procedure *plot_point* is a graphics display procedure written in EDEN.)

The EDEN definitions "_N **is** [LINE, ...]" and "_width **is** 800" are direct analogues of the corresponding DoNaLD definitions.

In the DoNaLD dialogue, the variables *width* and *length* act as two parameters of the room. The EDEN translation guarantees that the new shape of the room is automatically redisplayed when these parameters are changed. The triggered action associated with monitoring the constraint *desk_in_room* in DoNaLD behaves similarly.

There are clear advantages in adopting EDEN as the implementation language. If a traditional procedural language is used, the simplicity of the definitive model of the current state of dialogue is obscured by the procedural mechanisms that must be invoked to effect state transitions, and great care is needed to ensure that other procedures (such as display procedures) operate in consistent dialogue states. These problems can be ameliorated by using an object-oriented programming paradigm for information hiding, but the synchronisation of update and display must still be specified explicitly. An EDEN implementation considerably simplifies the prescription of display actions linked to the semantics of the underlying application. It can also treat simple constraint management in an appropriate manner viz as direct manipulation of the state of the dialogue by the computer. (As a simple illustration, maintenance of the constraint "x==y" can be conveniently handled by introducing actions ensuring that if the value of x - respectively y - changes between one dialogue state and the next, then the dialogue action "**if** (x!=y) **then** y=|x|" - respectively "**if** (x!=y) **then** x=|y|" - is performed. Here "|x|" denotes "the current value of the variable x".)

A potential problem with EDEN is that it supports methods of programming that may be powerful but can be obscure and difficult to analyse. There is the possibility of interference between actions, and of non-termination through recursive invocation of actions. Triggering of display procedures still means that reasoning about the current state of the display during an interaction entails knowledge of the history of a complex sequence of events. These concerns motivate a closer examination of the abstract machine model underlying EDEN.

2.3 The abstract machine model

Designing a medium for implementation is in effect designing an appropriate abstract machine code. The abstract machine model we have adopted solves the problem of representing the state of a general definitive dialogue by incorporating a low-level definitive notation as part of the machine code. In our case, the underlying algebra for this low-level notation is essentially based upon **lists** (as in Lisp) whose atoms are **integers**. The abstract machine can be then be viewed as having auxiliary definitive registers (DRs) that represent explicitly or implicitly defined **lists** and **integers**, and auxiliary machine instructions simulating the effect of redefinitions. The "hardware support" for the DRs includes a component that records the tree of dependencies between DRs, together with a mechanism whereby updating one DR automatically - ie as an indivisible action - selectively updates all dependent DRs. To implement a dialogue over an arbitrary definitive notation **D** on such a machine, the definitions in **D** are compiled into definitions in the low-level definitive notation in such a way that the updating of values of variables in **D** is carried out automatically. It will be convenient to refer to the "state of dialogue" within the abstract machine, as represented by the current definitions of the DRs, as the **internal dialogue state** (IDS), and the auxiliary machine instructions that update the IDS as the **internal transitions** (ITs).

As described, our machine model has the limitations of a pure definitive notation. It passively

supports a user dialogue consisting of a sequence of variable definitions by recording and maintaining the functional relationships between variable values, but has no independent capability for contributing to the dialogue (ie changing the dialogue state autonomously) or invoking "external actions" such as are required to display updated values, or declare error conditions.

To elaborate our model further, we must provide a way to program autonomous action contingent upon current dialogue state. Since activities such as constraint processing require responses to user dialogue actions that directly affect the dialogue state (eg actions that restore the table to its original position if it is placed outside the room), it must be possible to program our abstract machine to perform actions that can conditionally update the IDS. To achieve this, a trigger mechanism is introduced whereby updating (ie altering the value of) a DR conditionally schedules a sequence of machine instructions for execution. An action queue is included as a component of the machine for this purpose; "scheduling a sequence of instructions" is then interpreted as "pushing the given sequence of instructions in order into the action queue". Programming the abstract machine can be viewed as associating a (possibly empty) set of rules with each DR. A typical rule for the DR v is then denoted v ~> p, read "v triggers p", where p takes the form of a guarded command **if** g **then** s. Such a rule is interpreted:

when *the DR v is updated by an internal transition*
  if g *is true in the current internal dialogue state*
    then *schedule the execution of the sequence of machine code instructions* s.

To summarise: a typical IT comprises the redefinition of a DR that automatically causes all dependent registers to be updated, all triggers associated with updated registers to be activated, and all sequences of machine code instructions associated with true guards to be scheduled. No guarantee is given about the order in which instructions within distinct guarded commands are scheduled; the only assurance is that the next instruction will be executed only when there are no pending instructions for scheduling.

At this stage, the full exploitation of the trigger mechanism has not been seriously investigated; nor does this appear to be necessary in the present application. To reassure the reader that there is some prospect of effective use of queues of triggered actions, a few comments are in order. In the first place, interference between definitions in a dialogue is easily identified: it occurs only between two or more definitions of the same variable, or when a variable dependent upon a variable redefined in one definition appears in an evaluated expression in another. In the second place, subject to non-interference between the ITs triggered by an IT, the IDS reached after execution of all the triggered ITs will be unambiguously determined irrespective of the precise order of execution.

2.4 An abstract machine model for definitive programming

Although the system of DRs and associated hardware has been presented above as a way of extending a conventional Von Neumann machine, it is clearly sufficiently powerful in its own right to merit consideration as an independent machine model. We propose to adopt this novel abstract machine model as an appropriate computational model for supporting definitive programming systems (cf [Be86b], which describes a closely related model incorporating concurrency). As might be expected, this abstract machine has very much the same characteristics as the EDEN language. On the one hand, it can be readily programmed to perform complicated tasks. On the other hand - even though the elimination of conventional machine instructions from our model leads to a considerable simplification - the possibility of interference between triggered actions and of pathological non-terminating behaviour remains.

At first sight, programming in the definitive machine model is problematical. Without the support of the conventional procedural framework, it is not immediately clear how a satisfactory complete implementation of DoNaLD is possible, for example. There are two main areas in which the current implementation relies on conventional programming techniques: the parsing of DoNaLD for translation into EDEN, and the user-interface management.

The problem of writing a parser using a definitive machine model has yet to be seriously addressed, though there seem to be prospects for using suitably defined dialogue states to simulate the states of a parser, and EDEN actions to perform the appropriate "semantic actions". (For the time being, conventional syntax directed translation techniques will be used to convert DoNaLD input into EDEN.)

Our proposed solution to the problem of handling the display interface is to devise an appropriate definitive notation to describe the screen display. The triggered EDEN actions that formerly invoked procedures to display error messages, or to open a new context window, will then be replaced by dialogue actions in the display dialogue. The form of the definitive notation required is the subject of current research, but it is anticipated that there will be many advantages in adopting this approach. From the interface implementor's perspective, the format of the display interface can be partially described declaratively, and it will readily become possible to modify the display to reflect dynamically changing characteristics of the screen data. In effect, the current state of the screen display will be viewed as a system of definitions, rather than as a result of the cumulative effect of many procedure calls (cf [Fo87ab] and [Ha87]).

The reader who is apprehensive at the potential complexity of the abstract machine may take comfort from the fact that - as the EDEN fragments in §2.2 illustrate - its elementary use will suffice to implement most of the DoNaLD environment. It seems likely that constraint management is the only aspect that will call for more sophisticated use of the trigger mechanisms eg in using iteration for constraint satisfaction (cf [Bo86]).

§3 Retrospect and prospect

Though the practical development of definitive programming systems is as yet at an early stage of development, it is instructive to compare our approach with the object-oriented approach described in [Bo86]. The combination of declarative and procedural elements referred to in [Bo86] is itself a characteristic ingredient of programming with definitions. The role of SmallTalk as an implementation medium - in particular, the use of trigger mechanisms for the maintenance of constraints, and for animation - is in some respects similar to the use of the EDEN, as described in §2 above. (See also [Le87], where similar principles are applied in a different style.)

At present, our prototype system gives very limited support for constraints, but this does not reflect any difficulty of implementation other than that inherent in constraint manipulation as described in [Bo86] and [Ne85]. It should be possible to implement methods for constraint handling quite as sophisticated as these - by plagiarising [Bo86], and programming in EDEN rather than SmallTalk, for instance! Though there might be some virtue - perhaps eg some simplification - in translating techniques from an object-oriented to a definitive framework in this fashion, the development of definitive programming is not primarily aimed at simpler or more efficient implementation. The main advantage we anticipate in using the definitive programming paradigm lies rather in the conceptual grasp over the current state of an interactive dialogue that it provides [Be85]. At present, it is unclear how complex functional dependencies and the transparent acyclic system of functional dependencies provided by a pure definitive notation can be most effectively integrated to this end. In this connection, it should be noted that a simple functional relationship can sometimes obviate the need for a complex constraint. The parametrised definition of the door in Figure 1, for example, ensures that it is always within the room.

This research forms part of a broader programme concerned with the application of definitive principles to the design and implementation of CAD software; an area in which the problem of developing interfaces within which to integrate many different representations for a geometric object is particularly acute (cf [La87] and [Ta87]). The graphical notation DoNaLD is much simpler than the notations envisaged for CAD applications [Be87], but our use of the definitive programming paradigm outlined above is equally elementary. It will be of particular interest to

determine whether the use of functional programming concepts for specifying higher-order user-defined functions, or the use of symbolic manipulation to transform and simplify definitions, can be helpful in more sophisticated applications. The conspicuous absence of an explicit data base is also thought provoking.

Acknowledgements

We are much indebted to David Angier, Tim Bissell and Steve Hunt for contributions to the design of DoNaLD. We also wish to thank Mike Slade for helpful comments and suggestions.

References

[Am86] J Amsterdam, *Build a spreadsheet program*, BYTE July 1986, p97-108
[Be85] W M Beynon, *Definitive notations for interaction*, Proc hci'85, CUP 85
[Be87] W M Beynon, *Definitive principles for interactive graphics*, Proc NATO ASI: Theoretical Foundations of Computer Graphics and CAD, Il Ciocco, July 1987
[Be86a] W M Beynon, D Angier, T Bissell, S Hunt, *DoNaLD: a line drawing system based on definitive principles*, University of Warwick RR#86, 1986
[Be86b]W M Beynon, *The LSD notation for communicating systems*, University of Warwick RR#87, 1986
[Bo86] A Borning and R Duisberg, *Constraint-based tools for building user interfaces*, ACM Transactions on Graphics, Vol 5, No 4, October 1986, 345-374
[Ha87] P ten Hagen, R van Liere, *A model for graphical interaction*, Proc NATO ASI: Theoretical Foundations of Computer Graphics and CAD, Il Ciocco, July 1987
[Fo87a] J Foley, C Gibbs, W C Kim, S Kovacevic, *Formal specification and transformation of user computer interfaces*, Report GWU-IIST-87-10, Dept of Electrical Engineering and Computer Science, George Washington University, 1987
[Fo87b] J Foley, *Models and tools for the designing of user-computer interfaces*, Proc NATO ASI: Theoretical Foundations of Computer Graphics and CAD, Il Ciocco, July 1987
[La87] J Lansdown, *Graphics, Design and Artifical Intelligence*, Proc NATO ASI: Theoretical Foundations of Computer Graphics and CAD, Il Ciocco, July 1987
[Le87] C Lewis, *Using the NoPumpG Prototype*, University of Boulder, 1987
[Ne85] G Nelson, *Juno, a constraint-based graphics system*, SIGGRAPH '85, p235-243
[Ta87] T Takala, C D Woodward, *Industrial design based on geometric intentions*, Proc NATO ASI: Theoretical Foundations of Computer Graphics and CAD, Il Ciocco, July 1987
[Yu87] Y W Yung, *EDEN: an evaluator for definitive notations*, Final Year Project, Dept of Computer Science, University of Warwick , July 1987

```
int          width, length
point        centre, NW, NE, SW, SE
line         S, E, W, N1, N2

openshape door
within door {
    boolean     open
    int         size
    point       hinge , lock
    line        door
    open = true
    size = 200
    hinge = \NW + {10, 0}
    lock = if open then hinge - {0, size} else hinge+{size, 0}
    door = [hinge,lock]
}

width = 800
length = 800
centre = {500, 500}
.
.
N1= [NW, door/hinge]
N2= [door/hinge + {door/size, 0}, NE]
.
.
openshape desk
within desk {
    int         width , length
    point       centre, NW, NE, SW, SE
    line        N, S, E, W
    . . .
    width = 200
    length = 600
    . . .
    openshape drawer
    within drawer {
        int          width, length
        . . .
        length = \length div 4
        width = \width;
        . . .
    }
}
.
.
impose desk_in_room
desk_in_room = contains(room, desk)
monitor door_hits_desk
door_hits_desk = intersects(door, desk)
.
.
```

Figure 1: A sample DoNaLD dialogue