Scientific Visualization: Experiments and Observations

W M Beynon, Y P Yung, A J Cartwright, P J Horgan

Dept of Computer Science, University of Warwick, Coventry CV4 7AL

Introduction

In this paper, we consider issues of visualization from a programming perspective, proposing fundamental concepts that we need for successful visualization systems and suggesting general principles. We shall argue that there is a close parallel to be exploited between observation of physical phenomena and the interpretation of programs – in the real-world rather than abstract computational sense. In particular, we suggest that developing programming methods well-suited to scientific visualisation is intimately connected with the problem of making programs intelligible.

The paper is in four main sections. Section 1 considers scientific visualization in the abstract. Section 2 considers visualization in its relation to computational concerns. Sections 3 and 4 introduce an experimental system we have developed for visualization, indicating the principles on which it is based and some of its applications. The experiments and observations in our title refer both to the nature of our software prototyping and to the fundamental principles upon which our approach to programming is based.

1. Principles for Visualization

1.1. Visualization – a physicist's perspective

Scientific visualization is an issue of central interest and importance. A physical theory is most precisely formulated using abstract mathematical equations, but other ways of viewing equations guide physical intuition. Imagining field lines is essential for understanding electromagnetic theory, for instance. Feynman [12] §2-1 explains how the process of interpreting equations through such heuristic models distinguishes physicists from "mathematicians who study physics". Visualization is not a new subject – it is as old as physics.

A historical perspective on visualization is useful in understanding what computer techniques can contribute. The heuristic models of classical physics were based upon analogy with mechanical systems, as represented by the motion of waves, particles or gear-wheels, or by stresses in materials. The inadequacy of such models was first highlighted by attempts to reconcile two views of light, ambiguously associated with both wave and particle phenomena. This has had a major impact on the epistemology of physics.

In a series of essays [9], Bohr explains how the study of observations supplies a common framework for the mechanical models of classical physics and the abstract mathematical models of quantum theory (p.71):

> ... exhaustive description of atomic phenomena [...] makes use of a mathematical formalism [that] does not allow pictorial interpretation on accustomed lines, but aims directly at establishing relations between observations under well-defined conditions.

Bohr uses the word *phenomenon* to refer only to observations obtained under circumstances whose description includes an account of the whole experimental

arrangement. The paradoxical nature of light is resolved, since wave and particle models pertain to mutually exclusive experimental arrangements. Visualisation encompasses observations that cannot be directly perceived as visual, but include "registrations obtained by means of suitable amplification devices with irreversible functioning, such as penetration of electrons into emulsion [on a photographic plate]" [9] p73.

Bohr ([9] p.3) sees "the very essence of scientific explanation [as] the analysis of more complex phenomena into simpler ones". By focussing on phenomena, modern physics dispels the need for a universal mechanical model. There is no comprehensive model of 'what is actually happening' in electromagnetism. Each mode of visualization is appropriate for interpreting particular electrical effects. The modern status of visualization is summarised by two complementary quotations from Feynman [12]. On the one hand, no single picture of the electromagnetic field can replace the Maxwell equations:

> ... attempts to try to represent the electric field as the motion of some kind of gear wheels, or in terms of lines, or of stresses in some kind of material have used up more effort of physicists than it would have taken simply to get the right answers about electrodynamics. [12] §1-5

On the other hand, pictures of the electromagnetic field are quite essential:

> A physical understanding is a completely unmathematical, imprecise, and inexact thing, but absolutely necessary for a physicist. [12] §2-1

1.2. Visualization - a computer scientist's perspective

Computer models are not constrained to behave like conventional mechanical systems. A computer program can represent an exceedingly complicated pattern of state-transition behaviour that does not conform to familiar physical laws. In principle, this should enable us to make significant innovations in scientific visualization. In practice, a major obstacle is that computer programs are themselves hard to interpret with reference to their essential meaning.

There is a close parallel between the problems of representation in physics and computer science. An unambiguous mathematical description of the behaviour of a program is not sufficient to assist the external interpretation of behaviour. It specifies what the computation does – e.g. computing the value of (1+1+0.5+0.5)*50 as 150 – without indicating what the computation means – e.g. the price of wood for a picture frame, or the bus fare for a family of four. To paraphrase Feynman, we may say that the process of interpreting programs distinguishes the software engineer from the abstract computational theorist.

Better methods of developing programs with reference to their interpretation are needed for effective use of computers in visualization. The present status of computer programming in this respect is controversial. A particular concern is whether program specification within a conventional logical framework can effectively address the problems of relating a program to its external context. Smith [17], for instance, argues that a radically new framework for formalising programs is required. If the analogy between physical understanding of mathematical equations in physics and real-world interpretation of logical program specifications is appropriate, Feynman's reference to the completely unmathematical, imprecise and inexact nature of physical understanding points to a similar conclusion.

Object-oriented programming is a paradigm that was originally conceived with modelling of physical systems in mind. In Simula ([7] p61), programming is viewed as a restricted form of system description. This paper discusses a new programming paradigm that has similar philosophical roots to Simula, but is based upon the *phenomenon* rather than the *object* as fundamental concept. The abstractions that are required in this approach relate to modelling

observation – a perspective that is well-matched to modern physics. The shift of emphasis from *object* to *observation* is consistent with the epistemological developments in physics to which Bohr refers. A discussion of related issues in Russell ([16] Chap's XIII and IX) also endorses our choice of fundamental abstractions.

This paper will describe and illustrate how our approach to visualization is connected with the observation of physical phenomenon. We contend that our programming paradigm is well-suited to scientific visualization because of this connection. Related papers advance a complementary argument – that representing programs in terms of physical phenomena is a most appropriate way to deal with their real-world interpretation [5, 6].

2. Programming Principles for Visualization

2.1. Interpreting observation in computational terms

The fundamental concepts we adopt in programming relate to the observation of physical phenomena:

- a phenomenon is defined by a set of observations made within an experimental context in which a specified repertoire of state-changing actions is considered. For example, we may vary the temperature or pressure to which a volume of gas is subjected.
- different sets of simultaneous observations associated with a phenomenon typically exhibit functional dependencies; for instance, pressure, temperature and volume are related by Boyle's law, so that volume changes as a function of pressure and temperature.
- there are many different ways in which the same physical system can be observed and manipulated experimentally.

Some elaboration and qualification of these points is helpful.

In an experiment, we may be observing state-changes that are not directly under our control, as in measuring radio emissions from the sun's corona during a partial eclipse. This is reflected in the *agent-oriented* nature of our programming framework, in which we in general presume that there are state-changing agents other than the experimentor. Conventions have to be established for the observations that define phenomena. For instance, when observing a small heavy object falling, the effects of wind resistance may be discounted.

The concept of 'simultaneous observation' presumes intuitions about physical variables (e.g. the pressure, volume and temperature of a gas) whose values can be observed in different states within a single experimental context and are subject to change. It also presumes conventions about what constitutes indivisible propagation of change – for instance, we may ignore transient effects associated with abruptly changing the pressure to which a gas is subjected.

Our concept of dependency is not to be confused with an equational dependency such as is defined by global constraints on observations associated with a phenomenon. For instance, we shall interpret Boyle's law as an assertion about how *changes* in temperature or pressure indivisibly affect volume. Functional dependency is a more appropriate primitive concept than equational dependency for several reasons:

- the relationship between observations typically has the form 'if the parameters x_1, x_2, ... ,x_n are changed in a certain manner, then the parameter y changes subject to $y = f(x_1, x_2, \ldots, x_n)$'.

- a relationship between observations is typically uni-directional – there may be no independent method of changing the value of the parameter y
- it is not sufficient to think of y varying as a function of x *irrespective* of how x is changed. In hysteresis phenomena, for instance, increasing the magnetic field surrounding a metallic crystal will induce greater magnetisation, but reducing the field has no effect.

Our computational abstractions reflect a particular perspective on modelling a phenomenon. A phenomenon is conceived as associated with a set of states, as defined by comprehensive sets of simultaneous observations. Our model describes these states explicitly and expresses the way in which the values of particular physical parameters are linked in change. This method of modelling is appropriate even in a context where our knowledge of a phenomenon is incomplete and speculative. If our knowledge of a phenomena is comprehensive, it can be encapsulated using equational dependency. These two ways of regarding phenomena may be compared with procedural and declarative descriptions of programs, though the parallel is not exact (cf Smith [17]).

2.2. Fundamental principles in geometric representation

The simplest phenomena are those that involve observation and transformation of an object that has no autonomous behaviour. The experimentor is then responsible for deciding what parameters to observe and what changes of state to perform. Visualization of such phenomena involves specifying transformations of a geometric object that correspond to the observed behaviour of the physical object in a precise way. That is to say, observations of the physical object should be represented geometrically so as to reflect the way in which they are indivisibly linked with respect to atomic transformations. The term *geometric symbol* will be used to refer to such a representation.

Interpretation of a geometric symbol is enabled by a correspondence between geometric transformations of the symbol and actions performed on a physical object. As a familiar example, the icons of a desktop display typically represent accessible files. The nature of these files (e.g. whether they are executable or are data for a system application program) is indicated by the form of the icon. The deletion of a file is associated with removing an icon from the display. A subset of geometric transformations that can be performed on a displayed set of icons is in 1-1 correspondence with meaningful operations on the associated physical files, such as deletion, creation or conversion to a new format. The validity of the geometric interface depends upon the precise correspondence between geometric transformations of the display and meaningful operations on physical files.

By way of further illustration, consider Figure 1. Under one interpretation, the figure denotes a counter that is currently displaying "89". If the figure models the behaviour of the counter faithfully, there are two atomic transformations, one associated with incrementing the counter to display "90", the other with resetting the display to "00". Such semantically significant atomic transformations of a symbol will be called *interpretable.*

Basic operations on files are atomic in the sense that the user cannot observe intermediate stages in the process of file deletion or creation. They are also discrete, since partial existence of a file is meaningless. In general, physical objects undergo continuous transformations and indivisibility of changes in observed values is a concept that is properly invoked for arbitrarily small transformations. The same principles for the interpretation of geometric symbols still apply.

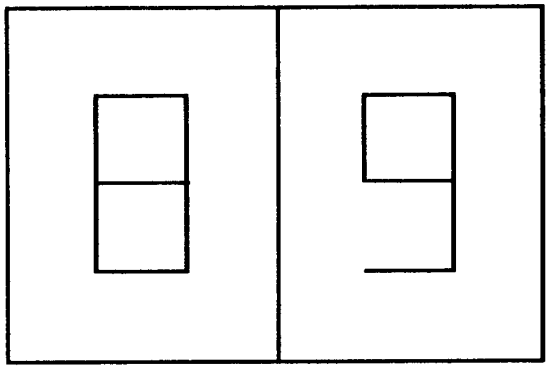


Figure 1

As an example, Figure 1 can also be interpreted as a plan for the furniture layout in a pair of adjacent rooms. The "8" symbol depicts a filing cabinet that is currently open, the "9" symbol the floor plan of a desk. The rooms have doors that are presently closed. In this case, the interpretable transformations of Figure 1 are of quite another kind, and correspond to actions an occupant might perform to change the state of a room, such as opening the door, moving a desk, or closing a filing cabinet. In this case, continuous transformations are involved.

There are many different phenomena associated with every physical object. Each phenomenon is determined by what observations and transformations are permitted. An experimental arrangement defines the privileges of the experimentor as observer and state-changing agent. The possible interpretations of a geometric symbol are determined in a analogous way by what we deem to be its atomic transformations (cf the association between the semantics of a geometric object and the transformations to which it can be subjected proposed by Klein [13]).

The class of interpretable transformations of a geometric symbol is determined by what agent is acting in the application and what privileges these agents have to change the system state. Conceiving different agents enables us to regard the same image as simultaneously having several alternative meanings. For example, reorganising the icons on a desktop display does not affect the status of the associated physical files, but is meaningful as a mode of presenting file information to the user.

Figure 1 likewise has many possible interpretations. The transformations of Figure 1 that correspond to updating a counter do not involve changing the presentation format for the digits. If on the other hand, the agent acting in the application is a graphical designer, it is necessary to consider transformations that modify the size and choice of representation for the digits "8" and "9", their relative position, or the colour of the background display. Similarly, when conceiving Figure 1 as an architectural plan, an architect might wish to modify the dimensions of the room, or relocate the door.

3. A programming paradigm for visualization

3.1. Basic concepts and techniques

The above discussion motivates the development of a method of representing a geometric object so that we can formally describe:

- the set of interpretable transformations that can be applied to it
- the set of agents that can perform such transformations upon it
- the conditions that constrain the performance of these transformations.

The drawing process is typically viewed as a means to an end: that of describing a static image (such as Figure 1) without constructing an abstract representation. From our perspective, a geometric symbol cannot be adequately represented by a static image; to reflect its semantics it is necessary to describe its geometric form together with appropriate protocols for transformation. This reflects the distinction between frozen figures as displayed in a document and dynamic figures as modelled in an interactive environment.

Traditional programming paradigms for graphics were not designed with the formal representation of transformations of geometric objects in mind. A procedural drawing package supplies many transformations that can be used to construct an image, but no framework in which to represent the "interpretable transformations of an image". Declarative methods (e.g. those based upon the use of constraints) give inadequate support to the concepts of state and transformation. A constraint is effectively an assertion about the form of a static object. Information about how an object is to be transformed when maintaining constraints is only supplied implicitly via a constraint-satisfaction strategy.

The fundamental technical problem to be addressed is the formal specification of the set of interpretable transformations of an image. In a conventional state-based computer model, such as a procedural graphics package describes, transformations of a geometric object can be specified informally, but there is no way to distinguish between interpretable and uninterpretable transformations. For example, a basic drawing package will provide a sequence of updating operations to modify the counter in Figure 1 so that the display "89" is transformed into "90", but this might take the form of introducing and deleting line segments so that the counter coincidentally displayed "88","80" and "90" in sequence. A method of expressing the indivisible nature of the transition from the state of displaying "89" to that of displaying "90" is required.

Our representations of geometric objects use variables to represent observed parameters of a physical object. Transformations of the physical object affect these observations in such a way that certain functional dependencies are respected. In our representation, the values of variables are defined by formulae that reflect these dependencies. In this way, a set of definitions of variables – or *script* – is used to represent observations of a physical object. In a script, each definition either specifies the value of a variable explicitly or defines it as the value of a formula referencing other variables (without cyclic definition). The redefinition of a single variable changes the values of all variables whose value is dependent upon it in a conceptually indivisible fashion. The principle is similar to that applied when updating the cells of a spreadsheet.

The nature of the formulae used in the definitions is determined by an underlying algebra of data types and operators over which expressions are evaluated. Listing 2 specifies Figure 1 as a room layout using the definitive (definition-based) notation DoNaLD – see below – in which the underlying algebra consists of points, lines and shapes comprising sets of points and lines. The basic transformations - such as that corresponding to opening the door - are described by the redefinition of a single variable. As an example, Listing 2 shows two files of DoNaLD definitions to describe the symbol "8" in Figure 1. In that on the right, the symbol "8" is transformed to "9" by incrementing the variable digit; on the left, the open filing cabinet represented by the symbol "8" is closed by setting the variable open to false. The protocols below prescribe the interpretable transformations.

```
openshape cabinet
within cabinet {
    int     width, length
    point   NW, NE, SW, SE
    line    N, S, E, W

    N = [NW, NE]
    S = [SW, SE]
    E = [NE, SE]
    W = [NW, SW]

    width, length = 300, 300

    SW = {100, 200}
    SE = SW + {width, 0}
    NW = SW + {0, length}
    NE = NW + {width, 0}

    openshape drawer
    within drawer {
        boolean     open
        int         length
        line        N, S, E, W

        length = if open then ~/length
                 else 0
        open = true

        N = [~/NW + {0, length},
             ~/NE + {0, length}]
        S = [~/NW. ~/NE]
        W = [~/NW + {0, length}, ~/NW]
        E = [~/NE + {0, length} , ~/NE]
    }
}

protocol {
    open -> open = false
    ! open ∧ ! locked -> open = true
    locked -> locked = false
    ! open ∧ ! locked -> locked = true
}
```

```
openshape led
within led {
    int         digit
    point       p1, p2, p3, p4, p5, p6
    line        L1, L2, L3, L4, L5, L6, L7
    boolean     on1, on2, on3, on4, on5,
                on6, on7

    digit = 8

    p1 = {100, 800}
    p2 = {100, 500}
    p3 = {100, 200}
    p4 = {400, 800}
    p5 = {400, 500}
    p6 = {400, 200}

    on1 = digit != 1 ∧ digit != 4
    on2 = digit != 0 ∧ digit != 1 ∧ digit != 7
    on3 = digit != 1 ∧ digit != 4 ∧ digit != 7
    on4 = (digit == 0 ∨ digit >= 4) ∧
            digit != 7
    on5 = digit == 0 ∨ digit == 2 ∨
            digit == 6 ∨ digit == 8
    on6 = digit != 5 ∧ digit != 6
    on7 = digit != 2

    l1 = if on1 then [p1, p4] else [p1, p1]
    l2 = if on2 then [p2, p5] else [p2, p2]
    l3 = if on3 then [p3, p6] else [p3, p3]
    l4 = if on4 then [p1, p2] else [p1, p1]
    l5 = if on5 then [p2, p3] else [p2, p2]
    l6 = if on6 then [p4, p5] else [p4, p4]
    l7 = if on7 then [p5, p6] else [p5, p5]
}

protocol {
    true -> digit = | digit | + 1
    true -> digit = 0
}
```

Listing 1

3.2. Software prototypes for visualization

Definitive notations are simple formal notations within which to formulate declarations and definitions of variables. Each variable may be defined explicitly by supplying a value of the declared type, or may be defined implicitly via a formula referencing other variables. Provided that the variables referenced directly or indirectly by the formula do not include the variable to be defined, the formula is meaningful and acceptable. For example if A, B, C are integer variables, then "A **is** B + C; C **is** B + 1; B **is** 1;" is a valid sequence of definitions. If this sequence is followed by "C **is** 1", then this definition of C will replace the previous definition. An attempt to redefine a variable that would lead to circular definition will be rejected by the system. This would apply, for instance, to the proposed

redefinition "C **is** A". This restriction upon definition ensures that the variables in a script always have values consistent with their definitions. Our visualisation techniques are applied to this consistent environment.

Each definitive notations is conceived with a mode of visualisation in mind. Each has its own set of data types and underlying algebra, appropriately chosen for the scope of applications. ARCA, DoNaLD and SCOUT are examples. ARCA was designed for the display and manipulation of combinatorial diagrams, DoNaLD for 2-dimensional line drawing and SCOUT for describing screen layout. To complement these special-purpose notations, the definitive language EDEN incorporates C-like data types and operators to facilitate more general applications and the implementation of other definitive notations.

The data types in the current implementation of the DoNaLD notation are integer, real, point, line, circle and shape. A shape is a set of points and lines. There are two kinds of variable of type shape. An openshape variable resembles a directory in a file system. It provides a context in which local declarations and definitions can be made. A shape variable is defined via an implicit formula. The following DoNaLD definition illustrates typical uses of these variables. A unit arrow is defined using an openshape variable and definitions of arrows to represent forces are defined in terms of this unit arrow using shape variables.

```
openshape arrow
within arrow {
        line arrowBody, arrowHead1, arrowHead2
        arrowBody    = [(0,0, {1, 0}]
        arrowHead1   = [{1,0}, {0.8, 0.2}]
        arrowHead2   = [{1,0}, {0.8, -0.2}]
}

shape wind
wind = rot(trans(scale(arrow, windF div 30), 200, 500), {0,0}, gradient)
```

Listing 2

The ARCA notation is oriented towards displaying and manipulating mathematical diagrams. Its data types are diagram, vertex, colour and integer. The kind of diagrams described by ARCA are combinatorial, as illustrated by Example 5.3. Vertices, colours and integers are used to specify vector, permutation and scalar information respectively.

The SCOUT notation is designed for screen layout. A SCOUT script defines the way in which sections of the screen are allocated for the display of diagrams and text defined in other notations. A SCOUT display is an ordered set of windows. A SCOUT window is best understood as a collection of area, content and attributes. There are different ways of defining areas for different kinds of windows. For a graphics window (such as a DoNaLD or an ARCA window), the area is defined as a box, and the required coordinate system is specified. For a text window, the area is defined as a list of boxes to allow multi-column text display. A typical example of a text window is that defining the 'off' button for the vehicle cruise controller status (Example 4.4):

```
point offBtnPos;
integer cruiseStts;

window crOffBtn = {
        type:       TEXT,
        string:     "OFF",
        frame:      ([offBtnPos, 1, 5]),
                /* a list of one box whose size is 1 row by 5 columns */
        bgcolor:    if cruiseStts == csOff then "white" else "black" endif,
        fgcolor:    if cruiseStts == csOff then "black" else "white" endif,
        alignment:  CENTRE,     /* put the word "OFF" in the centre of the box */
        border:     1       /* the border width of the box */
        sensitive:  ON
                /* will cause the system to generate a definition (keep track) of the
                   mouse state whenever there are mouse actions within this window */
};
```

Listing 3

A typical example of a graphics window is that defining the speedometer:

```
point origin;
integer width, height;

window speedometer = {
        type:       DONALD,
        box:        [origin, origin+{width, –height}],
        pict:       "SPEEDO",
                /* the DoNaLD picture named SPEEDO is to be displayed */
        border:     1,
        xmin:       –250,  /* only the region of the DoNaLD picture bounded   */
        xmax:       250,   /* by these four parameters will be displayed;     */
        ymin:       -250,  /* this region will be scaled to fit in the prescribed */
        ymax:       250    /* box                                             */
};
```

Listing 4

As the choice of data types and underlying algebra in these definition notations illustrates, definitive notations are designed to ensure a close correspondence between the definitive script and the screen display. This makes the interpretation of the script straight-forward.

Special-purpose notations are good for the end-user, but there are problems of implementation if we need to integrate them in a general programming framework. These problems are resolved in our system by having a common general-purpose definitive implementation language. All the definitions are translated to the definitive language EDEN and are subsequently maintained by the EDEN interpreter. EDEN uses common data types and operators such as can be found in conventional programming languages. For instance, the complex data types and operators in DoNaLD or ARCA can be emulated in EDEN using the list data type and user-defined functions. For details of the implementation method see [2]. The fact that all the definitive notations are translated into a common language that incorporates definitions means that a variable in one definitive notation can be

defined in terms of variables in another. Bridging definitions are introduced to convert a variable from one definitive notation. As a simple example, the DoNaLD declaration:

int I = ARCA int I

declares a DoNaLD integer I as an image in DoNaLD of an ARCA integer I. Such a declaration is still a definition in principle and falls within the scope of the definitive paradigm. In this way, specialisation of definitions is consistent with expressive power and breadth. Examples 4.3 [4] and 4.4 [3] illustrate how several definitive notations can be used effectively in conjunction.

4. Applications

In this section, we briefly review four applications of our visualization methods. Definitive principles are central to these applications, but each application has an experimental aspect aimed at understanding how best to exploit and develop the theme of this paper.

Example 4.1: The cube

Figure 2 depicts a cube specified in a definitive notation CADNO that is being developed for geometric modelling. The principles behind the design of CADNO are described in [1]. CADNO illustrates the way in which definitive notations support powerful reference mechanisms. This is exploited in the notation when abstract labels defined as values in a complex are interpreted as variable names in a carrier.

As currently implemented, the CADNO data types are the **complex**, the **carrier** and the **object**. An object is defined by combining combinatorial information of type complex with geometric information of type carrier. A **complex** is a set of faces of type **simplex**, each of which is a structured list of labels that specifies a combinatorial structure on abstract vertices. A **carrier** is a set of points specified using the conventions adopted for DoNaLD shapes. Object specification currently uses a default mode of realisation: faces are realised as rectilinear objects such as straight lines, triangles or quadrilaterals. In this process of realisation, a frame of reference can be specified for the object and for each of its faces. This is defined by an origin O and a triple of unit vectors X,Y,Z with standard defaults. (More complex modes of combining combinatorial and geometric information in objects are envisaged in the full design.)

In Figure 2, each of the 6 faces of the cube is defined as a 2-dimensional face bounded by 4 edges. In the complex faces, the vertices of the cube are assigned the abstract labels "/corners/a", "/corners/b" etc using the string concatenation operator (::). These labels reference points in the **carrier** corners. The object cubes is then realised by interpreting the combinatorial structure in the complex faces relative to the points in the carrier corners. The frames of reference of the sliding and hinged faces of the cubes are defined so that redefining the parameters pos and angle simulate their movement.

Example 4.2: A content addressable memory

Figure 3 illustrates the use of SCOUT, DoNaLD and Eden for visualization of a simple neural net. The network is based on an original model due to McClelland [14, 15]. The network represents information on individuals belonging to two gangs, the Jets and Sharks. Interrogating the network involves specifying initial activation levels for nodes and observing the activation pattern to which the network converges. For instance, to determine

the characteristics of Sam, we activate Sam's name unit. On convergence, the property units associated with Sam are activated.

In this example, visualization is used to animate a system with autonomous behaviour. The first phase of the visualization process involves specifying the form and layout of the symbols representing nodes. In Figure 3, this is specified using DoNaLD definitions. Each unit is represented by a box in which the activation level is indicated by a diagonal line. The same generic object is used to define for all units. Hidden units are depicted using dashed lines. In DoNaLD, all name units are defined within the context of a single **openshape** person that contains the parameters used to locate this pool of units. This mode of definition assists the design of the layout, making it possible to relocate the entire pool of units and the individual units within the pool. The designer can also assess whether the mode of visualising activation levels is effective by directly assigning activation levels.

Once the screen layout has been designed and specified in DoNaLD the only parameters that are changed in simulation of net activity are the activation levels of nodes. The mode of definition of the DoNaLD symbols automatically guarantees consistency between the screen display and the internal model. A feature of this approach to visualization is that the interpretable states and transformations of the neural net are precisely modelled prior to animation. At the same time, the screen layout can still be redesigned interactively and retrospectively simply by redefining appropriate parameters.

Animation involves simulating propagation of signals through the net. This is done by using an **action** mechanism in Eden, whereby changes to the values of variables triggers the execution of redefinitions. In the neural net, the trigger is a clock pulse that causes activation levels in all nodes to be recomputed according to a recipe based on those of neighbouring nodes and connection strengths. In this way, Eden actions animate the roles of independent agents that change the system state. These actions are complemented by actions implicitly specified in SCOUT that serve to define the user-interface. The SCOUT specification of the on/off button for the vehicle cruise controller in Listing 2 illustrates the essential principles.

Example 4.3: Simple 4-line arrangements in the plane

Figure 4 depicts an exercise in visualization of abstract mathematical models introduced in the study of combinatorial properties of planar arrangements. Full details of this are described elsewhere [4]. This illustrative example was developed using ARCA, DoNaLD and SCOUT in combination. When the relative distance between the endpoints of lines in the arrangement (top left) are interactively redefined, the combinatorial characteristics as depicted in the posets (top and bottom right) and in the Cayley diagram of the symmetric group S4 (bottom right) are changed accordingly.

There are several points of particular interest in this example. The program is almost entirely specified by a definitive script – only at one point is it necessary to use an Eden action to simulate a definition of a particularly subtle nature. The elements of the picture were first independently specified but were easy to integrate through the use of appropriate bridging definitions. At stages during the development of the final script, certain parameters needed to realise the diagrams appropriately had to be supplied interactively, but these could subsequently be specified by introducing appropriate definitions.

Example 4.4: A vehicle cruise control system

Figure 5 is a snapshot depicting an animation of a vehicle cruise control system. This visualization was devised to illustrate a simulation program that was itself developed using programming techniques based on definitive principles. Full details appear elsewhere [3].

This example makes use of techniques similar to those in Example 4.2. The interactions between agents, including the user, are too complex to specify directly using Eden actions in an *ad hoc* way. There are both discrete and analogue elements in the physical system and agents – including the user – act concurrently. A special-purpose notation (LSD), designed for defining agents and their state-changing privileges, was used to specify the system and the user-interface. LSD specifies how an agent can observe and affect the system state; it enables us to specify the privileges introduced in Figure 1, for instance. The use of LSD represents one way in which we have successfully enhanced definitive principles for representing state with a view to general-purpose programming.

Conclusions

Scientific visualization involves establishing a relationship between physical phenomena and computer models. This paper argues that new programming methods are appropriate for this purpose. The key principle is that of specifying transformations of computational and geometric objects so that they correspond precisely to observation of physical phenomena.

Our illustrative examples show that definitive scripts can play an important part in modelling observation. They also indicate that extensions of our methods are likely to be needed to represent functional dependencies between observations as they arise in general. Applications of constraint solving that involve automatic reconfiguration of functional dependencies represent one important area for future research (cf e.g. [10] Chapter 3).

References

[1] W M Beynon, A J Cartwright, *A definitive programming approach to the implementation of CAD softw*are, Int CAD Sys 2, Springer-Verlag 1989, 126-145

[2] W M Beynon, Y W Yung, *Implementing a Definitive Notation for Interactive Graphics*, New Trends in Computer Graphics, Springer-Verlag 1988, 456-468

[3] W M Beynon, I Bridge, Y P Yung, *Agent-oriented Modelling for a Vehicle Cruise Control System*, to appear in Proc ESDA, June 1992

[4] W M Beynon, Y P Yung, M D Atkinson, S R Bird, *Programming Principles for Visualization in mathematical Research*, in Proc CompuGraphics'91, Portugal, 1991

[5] W M Beynon, S B Russ, *The Interpretation of States: a New Foundation for Computation?* in Proc 4th PPIG, Jan 1992

[6] W M Beynon, Programming Principles for the Semantics of the Semantics of Programs, Research Report RR#205, Comp Sci Dept, University of Warwick, 1992

[7] G Birtwistle, O-J Dahl, B Myrhaug, K Nygaard, *SIMULA BEGIN*, 2nd Ed., Studentliteratur, Lund, Sweden, 1979

[8] G Booch, Object-Oriented Development, IEEE Trans Software Enginerring, SE-12(2), 1986, 211-221

[9] Niels Bohr, *Atomic Physics and Human Knowledge*, Science Editions, Inc., New York 1961

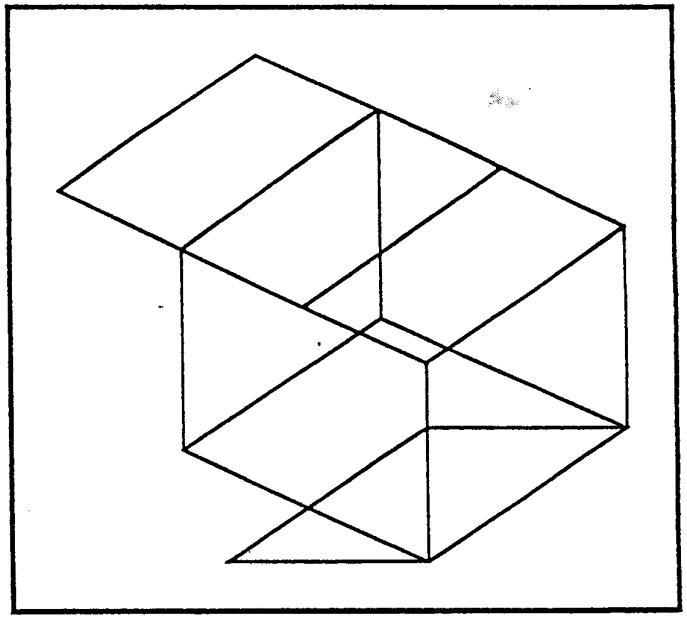
[10] S van Denneheuvel, *Constraint solving on database systems*, University of Amsterdam, 1991

[11] M S Deutsch, *Focusing Real-Time Systems Analysis on User Operations*, IEEE Software, Sept 1988, 39-50

[12] Richard Feynman, Robert Leighton, Matthew Sands, *The Feynman Lectures on Physics* Vol II, Addison-Wesley, Reading, Massachusetts 1964
[13] Felix Klein, *The "Erlanger program"*, 1872
[14] J L McClelland, *Retrieving General and Specific Knowledge from Stored Knowledge of Specifics*, Proc of 3rd Annual Conf of the Cog Sci Society, Berkeley, CA 1981
[15] D E Rumelhart, J L McClelland, *Parallel Distributed Programming: Vol 1 Foundations*, MIT Press 1986
[16] Bertrand Russell, *ABC of Relativity*, George Allen and Unwin, 1969
[17] B C Smith, *Two lessons of logic*, Comput Intell Vol 3, 1987, 214-218

complex faces

```
label l = '/corners/'
simplex
 front = [[l::'a',l::'b'],[l::'b',l::'c'],[l::'c',l::'d'],[l::'d',l::'a']]
 back = [[l::'e',l::'f'],[l::'f',l::'g'],[l::'g',l::'h'],[l::'h',l::'e']]
 left = [[l::'e',l::'a'],[l::'a',l::'d'],[l::'d',l::'h'],[l::'h',l::'e']]
 right = [[l::'b',l::'f'],[l::'f',l::'g'],[l::'g',l::'c'],[l::'c',l::'b']]
 sliding = [[l::'e',l::'f'],[l::'f',l::'b'],[l::'b',l::'a'],[l::'a',l::'e']]
 hinged = [[l::'c',l::'d'],[l::'d',l::'h'],[l::'h',l::'g'],[l::'g',l::'c']]
```



carrier corners

```
int size = 500
point
 z = {0,0,size}
 a = {0,0,0}
 b = {size,0,0}
 c = {size,–size,0}
 d = {size,0,0}
 e = a + z
 f = b + z
 g = c + z
 h = d + z
```

object cubes **on** faces

```
cubes_X = {cos(.25),sin(.2),0}
cubes_Y = {–sin(.25),cos(.2),0}
cubes_Z = {–0.5,0.5,0}
```

cubes_faces/sliding

```
sliding_O = {0,0,pos}
int pos = 200
```

cubes_faces/hinged

```
hinged_Z = {0,–sin(angle), cos(angle)}
real angle = 0.3
```

FIGURE 2 A CADNO specification of a box with faces that hinge and slide

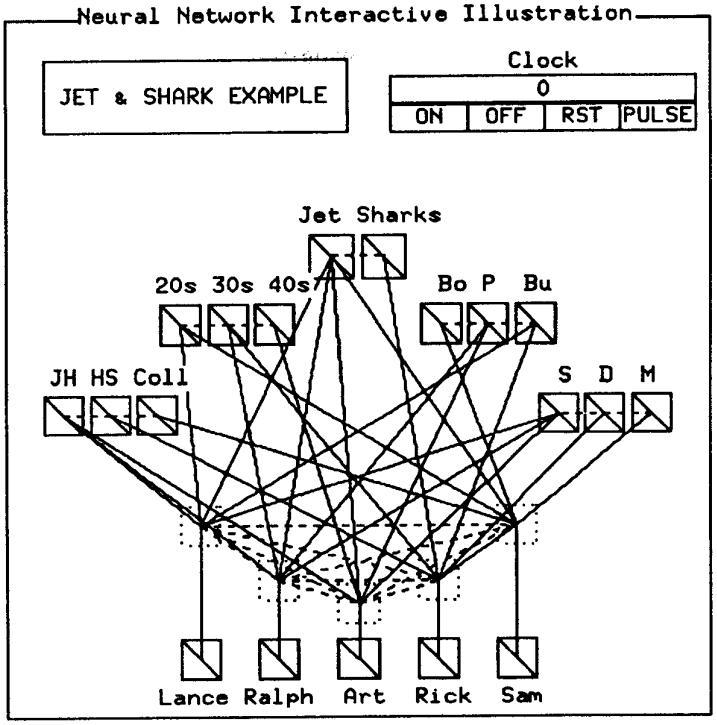


Figure 3.1: Initial Display of Example 4.2

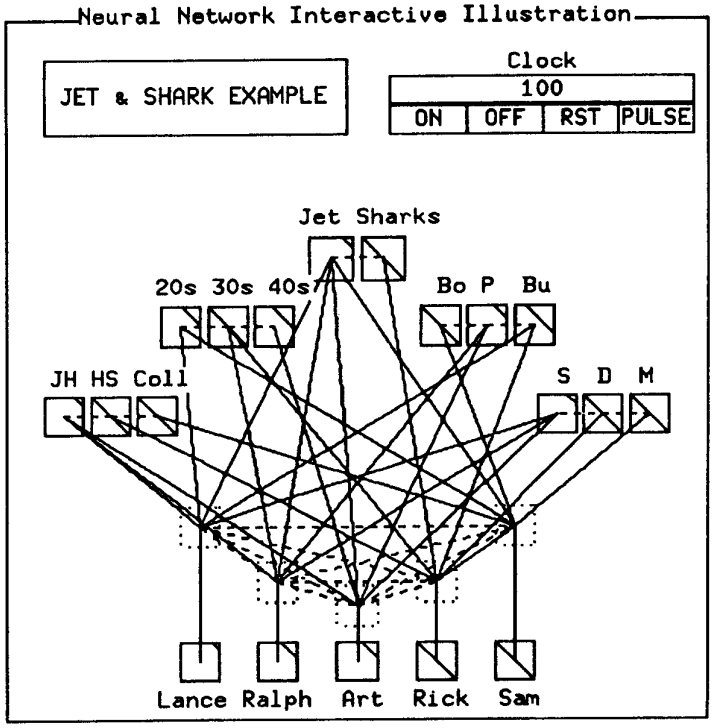


Figure 3.2: The Display after 100 Clock Cycles

Figure 3: A Content Addressable Memory

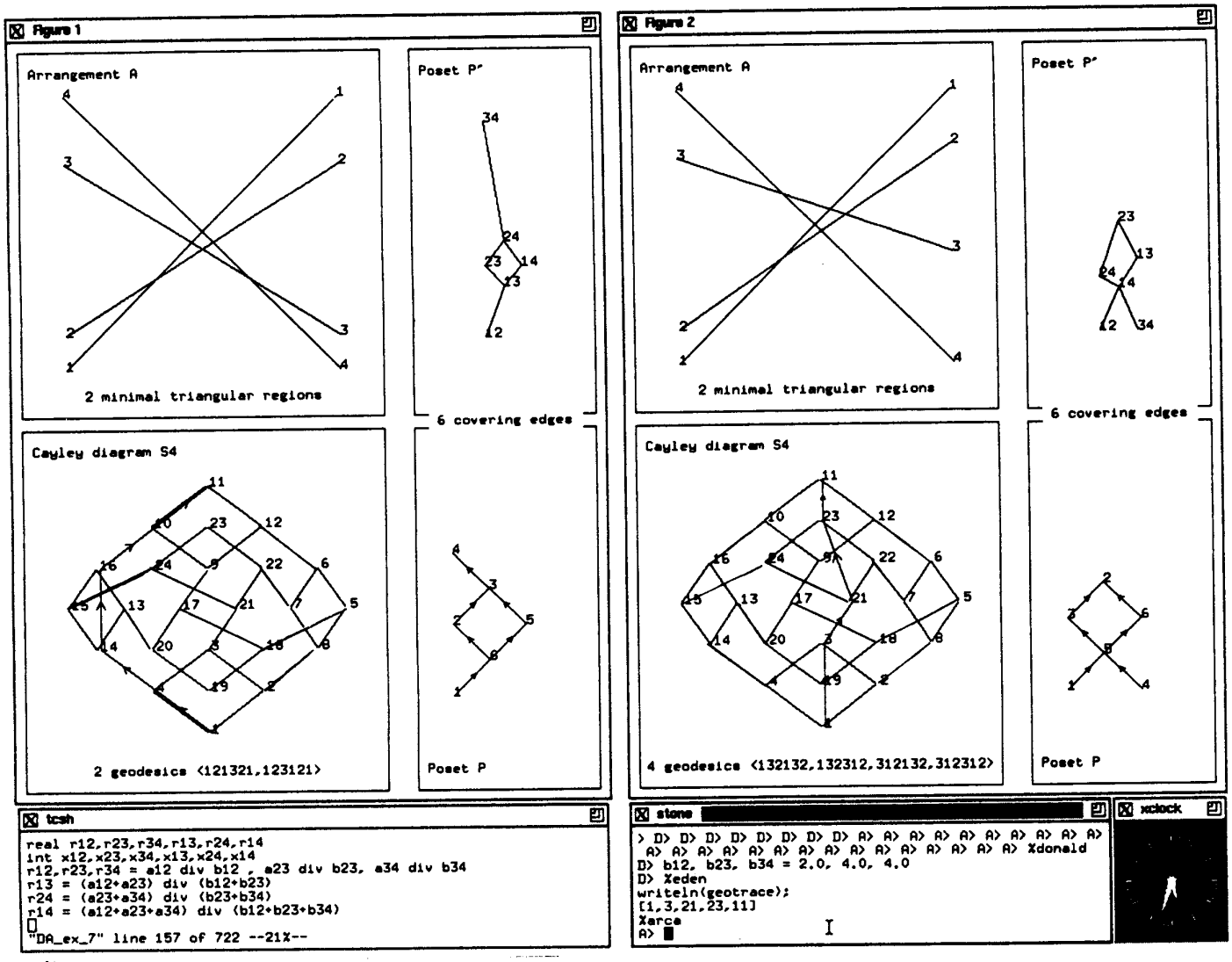


Figure 4: Visualization of simple planar arrangements of four lines

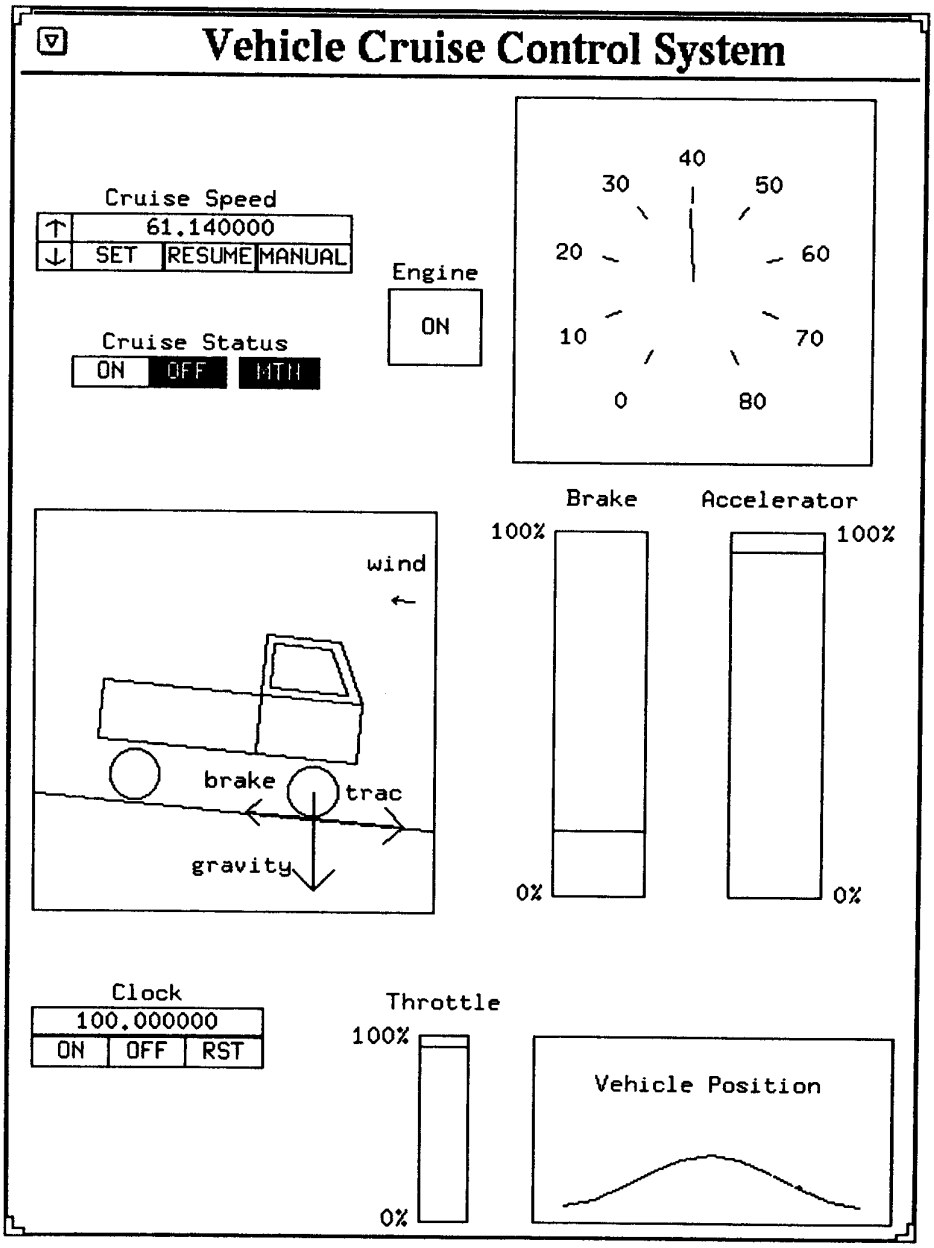


Figure 5: Sample Output of the Vehicle Cruise Control Simulation