A Model for Picture Structuring in Man-Machine Dialogs

Karl Soop
Mora, Sweden, 2018-11-24
Version 4.0

Contents

1. Introduction

A successful graphics application presents meaningful pictures to the user, and lets the user interact with the pictures in a natural way. Presumably the most natural way is *pointing*, which is also recognised by modern workstations (through mice, styli, or even the user's finger). The two operations: viewing and pointing, are fundamental to the very nature of pictures. They correspond to innate, almost automatic, capabilities of the human being, manifest early in life.

Can one base a formal computer-graphics model on these simple observations? Largely, yes, provided one includes elements from other recognised areas of system design. Two such areas contribute to the Model presented in this paper: structured and object-oriented design, with contributions from entity-relationship modelling and basic set theory.

Viewing and pointing — the two fundamental facets of a graphics dialog — obviously correspond to *presentation* ("output") and *interaction* ("input") in the system designer's world. These have traditionally been treated separately in graphics support and standards. The present Model brings them together in an object-based *formalised dialog*, including what might be called the "semantics" of interaction.

Evidently, picture and program *structuring* are closely related, and one can reap as much benefits from the technique in graphics as in program design. Yet, relatively few graphics systems provide more than one level of structure ("segments"), although most windowing systems provide an analogous multi-level structuring facility based on windows. The Model outlined in this paper offers indefinitely nested picture structuring as a natural companion to program structuring.

1.1 History

The ideas that led to the present model started emerging in my mind during early graphics work with IBM workstations, and found their first concrete form in the preparation of the Graphics Requirement Statement in 1976, of which I was a co-author. Many of its principles, among those related to the present topic, were adopted by the Hursley Laboratory, then the centre of graphics development in IBM, and found their way into several subsequent products.

In the ensuing years I was involved in graphics at many levels, providing an opportunity to further develop my original ideas. During my time as instructor of Graphics and class manager in IBM (ESRI, La Hulpe, Belgium), they were extensively tested and hammered in demos, student exercises, and hands-on training, using the tools I developed on top of the conceptual framework.

The latest result of these activities is a *Dialog Processor Model*, **DIPRO**. DIPRO can be regarded as a joint working name for the collection of ideas, theories, methodologies, and implementations, that emerged during these years, which are now further refined in the present package. Parts of DIPRO have been presented in Soop [1982, 1986, 1988a, 1988b].

1.2 What is DIPRO?

1. It is a **theory** describing the mechanism of a graphics dialog between a human and a computer (Section 5). Building on set theory, it is quite simple, even naïve, and on the surface perhaps not very new. But it has less trivial implications in terms of the functional split between a conventional processor executing the "traditional" application code, and a Dialog Processor executing the pictorial task of the application.

2. It is an **implementation** involving potentially a new *hardware design* for Dialog Processing (Section 6). The design evolves around a special-purpose associative memory, which was originally proposed by Symonds [1968]. DIPRO then becomes a unique vehicle for exploiting the power of *workstation intelligence* in a distributed processor network. It supports colour, high-resolution, and highly

interactive graphics. There are currently several working versions of DIPRO, emulated in software. One is described in the form of a user's guide in this document (Section 3.3).

3. It is a new application **building methodology**, providing the means for ultra-fast development of small to medium-sized applications with an interactive-graphics content (Section 4). The methodology is tightly connected with the theory of DIPRO, and extends well-known principles, such as Object Orientation and Structured Programming, into the new area of picture design.

1.3 What is DIPRO *not*?

1. It does not involve a new graphics **standard**, although it complies with the CGRM standard [ISO 1990]. It can coexist with standards like GKS as far as providing popular sets of graphic primitives. In this respect, though, DIPRO is closer to PHIGS [1985], since it is based on *structured graphics*. DIPRO is, in fact, open-ended as to the choice of graphic primitives, be they two- or three-dimensional.

2. It is not yet another graphics **function** or **class library**. DIPRO is intended to be implemented as an integral part of a laptop or a graphics workstation. The interface between a program and DIPRO is on the same level as a VDI (Virtual Device Interface), but is much more economical. It does not sustain the literally hundreds of function calls forced upon us by current standards and packages.

3. It is not a graphics **application**. Many graphics systems cater to specific application areas, such as Business Graphics or CAD. DIPRO is *neutral* in this respect. (In fact, DIPRO is probably less suited to Business Graphics, this being an application area with only trivial interactive contents.)

4. It is not yet another graphics **editor**. Paint tools and similar editors abound in the market and they all produce virtually static pictures, mainly for presentation purposes. DIPRO caters to *interactive* graphics. On the other hand, it is possible, and indeed a good idea, to develop a new graphics editor with DIPRO.

5. DIPRO is *Graphics*. It does not explicitly include Image Processing, Voice, or other **non-graphics techniques**, although it supports image-type primitives (”cell-array”, ”pixel-map”) as part of a structured picture, and may profitably coexist with these other techniques.

I would like to emphasise that most of the ideas in DIPRO are not new. They have been advanced and voiced in one form or another in numerous talks and articles by many authors. Rather, it is the consolidation of many concepts, bringing them to their logical limits, coupled with a few new ideas (especially in the Event/Response area), that leads to the power and utter simplicity of DIPRO.

2. Model Overview

2.1 Picture Structuring

Structured pictures are the basis for most modern graphics products, due to the power and flexibility they offer. This has been recognised by the PHIGS effort, which has led to an ISO standard [1985]. DIPRO also exploits structured pictures, but in a simpler and more pervasive manner than in PHIGS. The following is a résumé of the main ideas:

Looking first at structured *programming*, its foundation is no doubt the notion of a *function* or *subprogram*. Every programmer appreciates, and uses almost without thinking, the ability to call one function from another. The idea is that grouping program statements into a self-contained entity ensures a certain tightness when it comes to control, scoping, and interface.

For the designer of *pictures* — that is, the joint visual information of an application — this same mechanism is less obvious. Few graphics support systems give us the possibility to structure a picture like a program. Nevertheless, most graphic pictures are too complex to be comfortably developed in an immediate and direct fashion. An application designer would like to split the pictures into *subpictures*[1], describe them separately, and combine them in different ways. An example is shown in Fig. 2 (Section 2.6).

The analogy between subpictures and subprograms chiefly concerns structuring properties, although several other similarities are explored below. On the other hand, one should note that from a logical point of view a picture is not "executed" like a program. It therefore makes sense to regard the picture-handling system as at least conceptually distinct from the processor that executes the application code (the *Problem Processor*). As will be justified later, it is meaningful to characterise the former as a *Dialog Processor*.

When describing a subpicture, the designer makes use of basically three kinds of elements, or *properties*:

1. Graphic **primitives**, such as `polygon` and `text`, to be displayed in the subpicture. These properties describe the *What* of the subpicture.

2. Graphic **parameters**, such as the `position` or `colour` of the subpicture. These properties describe the *How*.

3. The **links** between the subpictures. These properties span the picture structure with a parent-child relationship, and may be said, in a topological sense, to describe the *Where*.

As will be seen later (Section 2.4), the designer is concerned also with a fourth kind of property:

4. The **response** when the user points at the subpicture. This property describes the *What If*.

Graphic primitives and parameters[2] evidently correspond to primitive data and types in the programming language, whereas the links correspond to function calls. Subpictures then become user-definable objects at the same level as functions, and we look for a suitable format of description.

Although the fastest way to "describe" a picture is no doubt just to draw it on the screen (using a smart picture editor), we are here concerned mainly with a formal description, a kind of source code, suitable as an interface to a programming system. It is perfectly possible to describe a picture in words and other syntactic elements, just like an algorithm is described by a programmer. Again, as in the case of an algorithm, one seeks a format that is independent of the platform on which the application runs. It therefore makes sense to call such a high-level source representation the *canonical form* of a subpicture (cf. the canonical form of APL functions).

---

[1]Other terms that have been used are "nested segments", "graphic entities", and "structures".

[2]Some, such as colour, are often called "attributes".

2.2 Canonical Picture Forms

A pictorial canonical form should provide a syntactic representation of the properties (1-3, and perhaps 4) listed above. The set of graphic primitives and parameters must be carefully designed; ideally it should be general-purpose, but might in practice depend somewhat on the target application area (e.g. supporting 2D or 3D). Its syntactic form is usually designed to match a familiar programming language.

At this point, we shall use only a few pictorial properties by way of example, and the syntax will be based on simple keywords in some imaginary scripting language (for a full set, see Section 3). Let us consider the example of a subpicture `OSCAR` displaying a tetragon and some text, positioned and coloured as to the programmer's intent:

```
OSCAR
   position 6j8,
   colour red,
   figure 0 10 7j6 -3j6 0,
   text "hi there";
```

This example assumes a simple punctuation syntax, where 2D coordinates are expressed as *`xjy`* (akin to a complex-number format), but an API using a different notation may be desirable, depending on the context.

If the subpicture should link to (contain, include) another subpicture `FREDDY`, defined elsewhere, one adds the line:

```
   link FREDDY,
```

We shall say that `OSCAR` in this example is a *parent* subpicture of `FREDDY`, and conversely that `FREDDY` is a *child* subpicture of `OSCAR`.

An important aspect of the canonical form underlines its difference from that of an algorithm: the order between the properties is of no importance. In other words, we assume that the picture is the same whether the `text` is displayed before or after the `figure` (as is well known, this is far from always the case with most technologies). Similarly, the fact that the subpicture will be positioned at <6, 8> remains valid for all its contents, thus this line can be stated anywhere in the canonical form. The same is true for the colour; both the tetragon and the text will be red. The canonical form is therefore *declarative*, analogous to a class or type declaration in a programming language.

All parameters are local in the sense that they pertain to the subpicture where they are stated (cf. local objects in functions or classes). Nevertheless, parameters must be expressed with respect to *some* reference system. We shall assume, as do most advanced graphics support systems, that many parameters are expressed relative to a higher-level subpicture in the structure. Thus the position <6, 8> is relative to that of a parent subpicture (with a `link` to `OSCAR`). Such "propagation" or "concatenation" of parameters is a well-known mechanism in computer graphics. One may observe that all the usual parameters, such as `position`, `orientation`, `scale`, `window`, `colour`, `font`, `texture`, as well as `visibility` and `detectability` of the subpicture, are good candidates for similar propagation.

2.3 The Relational vs. the Object View

Another important aspect of the canonical form is that each property can be split into two parts: the *type* (keyword) of the primitive or parameter, and its *value*. An example is the line "`colour red`" above, where the type is `colour` and the value is `red`. The value part can be expressed by constants, but since the subpictures are not executed (i.e. not handled by the Problem Processor; Section 2.1), expressions must be disallowed in canonical forms.

The duality of picture properties points to *binary relations* as a means of expressing the information. Each graphic property type (primitive, parameter, or link) may then be considered an innate relation of the Dialog Processor. One regards, for example, `position` as a binary relation between a subpicture (say, `OSCAR`) and a value (say, `6j8`). In a similar way, `link` is a relation between `OSCAR` and `FREDDY`, or perhaps

between `OSCAR` and a set of several children. Other examples are given by Symonds [1968], Sharman [1979], and Palermo [1979] (cf. also Schauer [1983]).

Following the relational view of the subpicture structure, the programmer may access a subpicture property through a form of the relational *join* operation, here denoted by the symbol "#". Thus, given a suitable API, the application program may *set* the colour property with the statement:

```
colour#OSCAR = yellow;
```

where `yellow` is an object of the appropriate type; and *fetch* it with:

```
C = colour#OSCAR;
```

Note that, since these statements are part of the application program (Problem Processor), expressions are, of course, allowed and normally used in this case:

```
colour#OSCAR = findColour(1,x);
```

An alternative, and largely equivalent, view of the type-value duality comes from regarding the subpicture as an *object*. Each pictorial relation or type then becomes a *property* of the subpicture class, and its value becomes part of the *state* of the individual subpicture object. This view is the basis for the formal theory of DIPRO (Section 5), whereas the relational view is more relevant to the implementation (Section 6).

2.4 The Dialog

The graphics model we have just sketched is entirely static. It permits arbitrarily complex pictures to be built, using a canonical form analogous to that of an algorithm. The form may be accepted (compiled, etc.) by a DIPRO implementation, but the resulting subpictures are established in a memory space other than that of the Problem Processor. This workspace belongs to a Dialog Processor, and the only API we have with the picture consists of a *join* operator (Section 2.3). We do not even have the means to start the Dialog Processor in order to display our picture.

The aim of this and the following sections is to see how the picture begins to live; how it is displayed, how the user interacts with it, and how it changes according to the programmer's intent.

The basic vehicle is a model of a graphics dialog between a user and a computer. We begin by considering the *user's* view of the dialog, as outlined in the Introduction. These two steps are repeatedly cycled through during an application session:

a. **View** the picture on a display device (which might also be hard-copying). In some cases the picture changes without user interaction (*animation*).

b. **Interact** with the picture by pointing at it, usually after considering a number of choices (assuming an interactive display device). Pointing takes place via some instrument (such as a mouse-driven cursor), or with some devices directly by the user's finger.

This is translated into model terms by considering the corresponding mechanism inside the Dialog and Problem Processors. Four phases are defined, which are repeatedly cycled through:

1. **Presentation**. A picture is presented on a display device. The model does not exclude several devices, each one with its picture. The picture is generally composed of subpictures to any required depth.

2. **Access** (of an event). The dialog halts here pending the arrival of an *event*. Normally, in an interactive application, the event is caused by the user pointing at the picture. But the model also allows for other event sources, such as a keyboard, timer, an external process (e.g. a sensor), or even the application program itself (*posting*), which may be perceived by the user as animation.

3. **Correlation** (of the event). Here the system attempts to determine what the user pointed at in the picture. In a hierarchical structure, this information is uniquely given by a *path* through the structure, consisting of the names of subpictures involved, including the primitive instance detected by the pointing.

4. **Response** (to the event). Some part of the application is *executed* following the interaction. The code to be invoked depends (a) on the previous Correlation, as well as (b) on the application state. Typically the response consists of any traditional application processing, such as calculations and data-base accesses. Of special interest is that the response also often modifies the picture to be presented in the following phase 1.

This model is sufficiently general to encompass the dialogs in practically all graphics applications.

2.5 Functional Distribution

We note that the first three phases listed above are concerned only with the processing of pictures, and might therefore be executed by the Dialog Processor. Only phase 4 utilises conventional program code, and must therefore be executed by the Problem Processor. Consequently DIPRO assumes a split of execution support into two communicating processors (Fig 1).

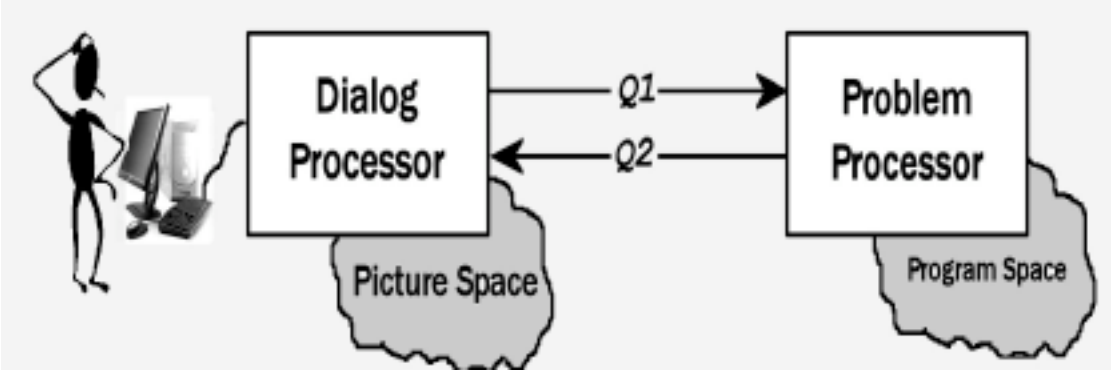


Fig. 1 Functional Split

The Dialog Processor continually executes the first three phases of the dialog, displaying the pictures and correlating the events with respect to them. It possesses a *Picture Space*, analogous to the program space of the Problem Processor, where all subpictures belonging to the application are stored. The two processors are asynchronous, but communicate through a queuing mechanism explained next.

2.6 Response Logic

In each cycle of the dialog, at the end of phase 3 (Section 2.4), the Dialog Processor must make the result of its correlation available to the Problem Processor to allow the latter to respond. The format is a path, composed of the names of subpictures detected in the interaction. For example, if the user pointed at line segment number 5 in subpicture `WHEEL2` in subpicture `AUTO`, which in its turn is placed in a `DESIGN` area on the screen `CONSTRUC`, the path would consist of these names:

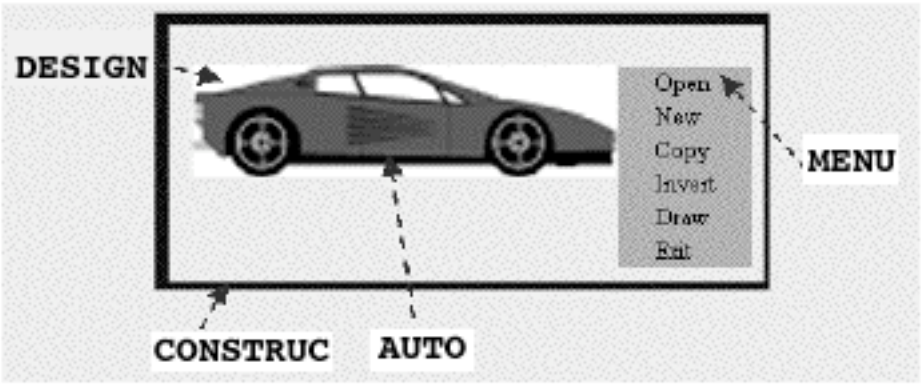
```
CONSTRUC DESIGN AUTO WHEEL2 5
```

where each name in the path denotes a parent subpicture of the next one (see Fig. 2). The path is communicated to the Problem Processor through a queue, *`Q1`*.

`Q1` is used by the Response (phase 4) of the dialog cycle, so it may be called a *response queue*. Because of its fundamental role in the model, it should preferably be accessed through a *system variable*, say `RQ`, in the API.

It is convenient to define the queuing protocol in such a way that each reference to `RQ` by the Problem Processor yields the next name in the Response Queue. This allows the application program to perform a logical selection based on each path element in turn. Referring to the example above, when element `DESIGN` is found, the function that handles all general manipulations of the design is invoked — one may

give this function the same name `DESIGN` (or some name derived from it) in order to automate its invocation. `DESIGN` gains control, picks the next element through `RQ`, finds it is `AUTO`, and invokes the function that handles the `AUTO`. Finally, `AUTO` calls `WHEEL2` which executes the intended manipulation, for instance erasing edge 5 in the wheel.



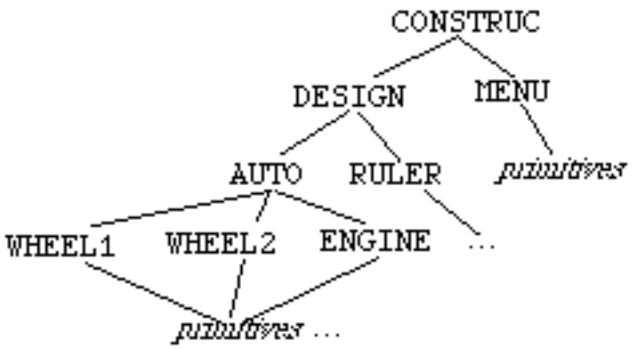


Fig. 2 Example of Program Structure = Picture Structure.
Each arc in the structure represents a parent-child relationship between two subpictures.

One may observe that this kind of logic in a well-structured application follows more or less the picture structure. In general, there are several possible choices at each node. Thus, at node `CONSTRUC` the user may point at either the `DESIGN` area or, say, at a `MENU`. Similarly, at node `AUTO`, the vehicle in question consists of several parts other than `WHEEL2`. One speaks about a *fan-out* logic, where the invocation target depends, at each node, upon the next element in `RQ` (but on no other element). An application wishing to use this logic efficiently would contain a utility function to perform the fan-out (via e.g. a `switch` statement or a polymorphic function call).

The queuing mechanism also yields an elegant solution to the synchronisation of the processors. After having performed all responses, the Problem Processor must somehow signal the Dialog Processor to start a new cycle. To do this, it just references the system variable `RQ` one more time. Queue *`Q1`* is now empty, but the Dialog Processor is nevertheless obliged to return a result. The only way it can satisfy the demand is to re-execute the Presentation, Access, and Correlation phases.

2.7 Picture Updates

As previously noted, an important task for the Response phase is to update the subpictures (if it did not, the user would always face the same picture on the screen). Since these objects are located in the picture space

of the Dialog Processor, the request is directed in the sense opposite to *Q1*. The communication can be implemented through a second queue *Q2*, the *request queue*.

For example, using the syntax suggested in Section 2.3, in order to update the position of subpicture OSCAR, the program would execute the statement:

```
pos#OSCAR = 5j6;
```

(or equivalent in the language used). We conclude that any reference or assignment to one of the subpicture properties gives rise to traffic between the two processors.

The Problem Processor enqueues the update requests on *Q2*, expressed as triplets:

<*property*, *subpicture*, *value*>

(in the example as <pos, OSCAR, 5j6>) and handles the required protocol. *Q2* is then received by the Dialog Processor, which goes on to service the request. Reference requests are expressed as pairs:

<*property*, *subpicture*>

and the resulting value (or set of values) is sent back to the Problem Processor on the same channel as used for *Q2*.

3. Pictorial Properties

3.1 General Considerations

Different implementations of DIPRO may support a different repertoire of subpicture properties. The choice is intimately connected with the capabilities of the workstation, and may also depend to some extent upon the intended application area. For example, due to different primitives (3D), one might expect a DIPRO implementation for CAD not to have the same set as one aimed at cartography. Notwithstanding, there are certain properties that are more important than others, and without which it would be difficult to imagine any implementation of DIPRO.

For example, the fact that a subpicture `S` is part of a hierarchical structure is embodied in the `link` property (Section 2.2). The `link` property is therefore mandatory in any DIPRO implementation. If `S` has the property `link T`, then `T` is a *child* of `S`, and is presented and correlated as part of `S`; conversely `S` is a *parent* of `T`. Note that nothing prevents other subpictures to have links to `T`, the net effect being that several *instances* of `T` are shown during the presentation phase. The only restriction is that loops must be avoided in the structure (which becomes effectively a "re-entrant tree" or an "acyclic, directed graph").

As another example, consider the parameter `colour blue.` This property would typically specify the colour to be used for primitives in the subpicture, at least as a default, or possibly in some other sense to be precisely defined by the implementation.

The role of `colour` and `link` as properties is not difficult to grasp, but the situation is less obvious for primitives. But since primitives can be regarded as subpictures at the lowest level, i.e. as leaves of the structural tree (cf. Fig. 2), one may regard them as special links. Thus a *primitive property* simply states that the subpicture contains an instance of this primitive, establishing a link with what is often implemented as a *generator* in the hardware. For example, if the hardware supports circles, then the Dialog Processor would provide a property `circle`, which may be used by an update request to "link" `S` to the circle generator:

```
circle#S = 35;
```

The effect would be that a circle of radius 35 is shown as part of `S`.

In summary, subpicture properties are used uniformly to express *links*, *primitives*, and *parameters*.

3.2 Discussion of Parameters

A parameter may affect the *geometry* (spatial appearance) or the *style* (cosmetic appearance), or perhaps some other aspect, of the subpicture for which it is specified. Apart from the *local* value, specified in a subpicture `S`, one may speak about the *effective* value of a parameter, meaning the value actually used when an instance of `S` is presented or correlated (cf. Section 2.4).

3.2.1 Parameter Propagation

`colour` is an example of a parameter that affects the style of the subpicture. Since a subpicture in one sense "contains" its children, the colour would be expected to somehow *propagate* downwards in the structure. Thus, if `S` has links to `T` and `U`, one would expect `T` and `U` to be red if `S` is. In this case, the effective colour of the children `T` and `U` is red, even though no local colour was specified for the latter. What happens if `T` also has a local `colour` parameter, specifying yellow? If the DIPRO implementation supports colour blending, `T` may be shown in orange. This is an example of parameter *combination*. One can say that the colour parameter in this example is *relative*, as opposed to an *absolute* parameter, which would override the value propagated from the parent without combining with it. Note that this distinction is not important, as an absolute parameter may be regarded as a special case of a relative one (where the combination ignores the propagated value).

3.2.2 Geometric Parameters

Geometric parameters almost invariably propagate by combination ("concatenated transformations") and are therefore *relative* in systems that support structured graphics. Thus, if the local position of a subpicture `s` is given by a parameter `pos`, all its children are translated (shifted) along with `s` whenever its `pos` is updated. We may say that position combines under vector addition.

In some systems common geometric parameters are given jointly by transformation matrices (often using homogenous coordinates); however, in the examples in the following section, they are explicitly and independently stated as *position*, *orientation*, and *scaling*. The reason is that we would like to show an API that caters to a more casual application developer. After all, a matrix is a highly user-hostile format of specification. Orientation (rotation) and scaling will then propagate and combine under vector and scalar multiplication, respectively.

3.2.3 Windowing

A `window` parameter is of crucial importance in any graphics system. *Windows* may be used for clipping or for mapping (onto viewports); in the example below, only the former technique is provided, since the other geometric parameters may be used for mapping. Moreover, in a hierarchical structure, an explicit mapping parameter (`viewport`) is not needed, since a window specified for a parent subpicture automatically acts as a viewport in its child. Finally, viewports, in the strict sense as a physical area on the display surface, only occur at the highest level of the hierarchy. In DIPRO the root subpicture of the structure represents the physical device itself, so the Dialog Processor will map its `window` directly onto the application window (client area or *canvas*).

In general, `window` is therefore a parameter that limits the subpicture to within a given (usually but not necessarily rectangular) area, causing material outside the window frame to be clipped. The window of a subpicture is itself clipped by the effective window of its parent. Thus, the parameter is relative and combines under set intersection.

Properties may be provided to facilitate interaction with the picture, with but little effect on the presentation. An example in the following section is `locate`, a primitive embodying the *locator* mode of "graphic input" in some current standards. By default the `locate` primitive is not shown on presentation, but covers the effective `window` with invisible pixels that the user can point at. Similarly, a pad of *function keys* may be regarded as a (very coarse) pixel matrix that can be pointed at, even though it is not physically located on the display screen.

3.2.4 Dynamic Parameters

In many graphics systems `visibility` and `detectability` are parameters (or "attributes") that can be set on or off. The example in the next section expands the idea of an on/off state to handle any state named by the programmer. The state itself is then set by a third parameter `state`, which can be said to embody the actual object state. This provides a very powerful and flexible mechanism to regulate the dynamics of a picture based on *state networking*, a well-known discipline in application development (Section 4).

To this group of *dynamic* parameters, one might add `hiliting`, used to emphasise the presentation of a subpicture (with states `selected`, `unavailable`, `normal`, etc.).

3.3 An Example Set

This section lists a full set of pictorial properties by way of example. The list serves partly to concretise the ideas developed so far, partly as a feasibility check on the proposed implementation (Section 6). At the same time, this section may serve as a User's Manual for the software-emulated implementation provided as a companion to this document. Unless stated otherwise, to *point* in this example means to click with the left mouse-button.

The Dialog Processor, described in the example, supports general-purpose, medium-level, two-dimensional, interactive, colour graphics. In addition, an experimental 3D extension is included (Section 3.4).

3.3.1 Syntax

Subpicture properties are specified in a canonical form (Section 2.2). Each statement is headed by the name of the subpicture. One or more properties may be given, separated by commas. The statement ends with a semicolon. In other words, a statement is:

name property, ...;

where *name* designates a subpicture, and an ellipsis (...) is used here and later to denote indefinite repetition of the previous syntactic unit. If the subpicture is undefined (i.e. *name* has never been used), it is automatically created.[3]

A *property* is:

typename value

The effect is to set the property *typename* to *value* in the subpicture *name*. The *typename* is a member of a set of implementation-defined keywords. The *value* may be composite, consisting of several components (see examples in the following sections). It may also be omitted (see below).

Setting a property that is already set in the subpicture *replaces* it; for example, in:

```
oscar colour red, colour green;
```

the effect will be the same as:

```
oscar colour green;
```

However, for *links* and *primitives*, you may *add* the property without replacement with a '+' syntax.[4] Thus:

```
oscar link fred, +link beata;
```

has the same effect as:

```
oscar link fred beata;
```

and

```
oscar +box 6;
```

will add the box to any other boxes already present in `oscar`.

If *value* is the reserved word `nil`, the effect is to *remove* the property from the subpicture (if present).

If *value* is omitted, the property is not set but *queried*, causing DIPRO to return the value of the property. If the property has not been set in the subpicture, `nil` is returned.

The following syntax is used here and in subsequent sections:

- *name* is an unquoted character string forming a bona-fide identifier in the usual programming sense. It usually denotes a programmer-invented subpicture or state name.
  Examples: `oscar, R1234, a_B.`
- *point* is a 2D coordinate, expressed as *x*`j`*y* (it may be regarded as a complex number with *x* the real and *y* the imaginary component), or only as *x*, in which case *y* is 0 unless stated otherwise.
  Examples: `5j8, -1j1, 0j-1, 0j-1.5537, 4, 4.8`.
- *string* is a character string enclosed in double-quotes. Examples: `"hi there", "".`

[3]In a theoretical sense subpictures are never created or destroyed, only their properties are (Section 5.1.5).

[4]Without this rule, the programmer would have to repeat all previous links, say, just to add a new link. The cost is that an additional syntactic element is needed to remove a link (Section 3.3.7).

- *colcode* is either a triplet of integers forming a colour code, or a mnemonic. The integers denote in order the intensities of the red, green, and blue components in the range [0, 255]. The following mnemonics are used as a short-cut for the most common *colcode*s:

```
black       darkgrey        palegrey
red         darkred         palered
green       darkgreen       palegreen
blue        darkblue        paleblue
pink        darkpink        palepink
cyan        darkcyan        palecyan
yellow      darkyellow      paleyellow
white
```

Examples with mnemonics:

```
255 0 255 (pink)
255 255 255 (white)
0 0 0 (black)
0 105 8 (sort of grey-green hue without a mnemonic)
```

In the following the syntax of each supported property type is stated, then a general description in terms of its effect on the subpicture `S` for which it is specified, followed by any dependencies on other properties. The term *local* pertains to a value specified in `S` itself, and *effective* to the value actually used in the presentation of (an instance of) `S`.

3.3.2 Primitives

Primitives are described in terms of how they are presented and how they react on interaction. The latter includes what the primitive contributes to the Response Queue (section 2.6) when the user points at it. In some cases, the conditions for detecting a primitive by pointing are also given.

`figure` *pointlist* **`/ ...`**

where *pointlist* is a list of *points* separated by blanks.

Each *pointlist* is presented as an open polygon passing through the specified points, expressed in the local system. A "polygon" of one point is presented as a dot.

On pointing, a list of 0-based indices is returned, identifying the detected polygon edges (note that it is possible to point simultaneously at several edges).

Examples:		Comments:
`figure`	`3j3 4j-10 0j5 3j3`	Triangle
`figure`	`10.8`	Dot on *x*-axis
`figure`	`0 -10 / 0 3j5 / 0 3j-5.1`	Y-shaped figure
`figure`	`10/0/-10`	Pin-point figure (3 dots)

Remarks: This is the traditional line-set primitive ("polyline").

`box` *point* **`...`**

Presented as one or more rectangular boxes. Each box is centred at the local origin, is parallel to the local axes, and has a vertex at the specified point. If the *y* component of *point* is omitted, the *x* component is assumed (the box is square).

On pointing, the interior of the box is detected, and the integer pair `0 1` is returned.

Examples:	Comments:
`box 3j4`	Box through point <3, 4> (will be 6 wide, 8 high)
`box -3j4`	*idem*
`box 2j4 5j5 10j9.7`	3 concentric boxes
`box 10`	Square
`box 10j10`	*idem*
`box 10j0`	Very flat box (straight line)
`box 2j10 10j2`	Cross made of 2 boxes

Remarks: This and the following primitive epitomise a large class of special-purpose primitives that are often included in implementations. As defined here, the primitive must be centred by the `pos` parameter (below). The composite value returned on pointing is compatible with the 3D case (Section 3.4.1).

`circle` *point* **`...`**

Presented as one or more concentric circular arcs. Each arc extends anti-clockwise from the local positive *x*-axis to the specified point, and is centred at the local origin. Ellipses may be produced by rectangular scaling (see the `scale` parameter below). If a point on the positive *x*-axis is specified, a whole circle is presented.

On pointing, the perimeter of the arc is detected, and its coordinate in the local system is returned, but normalised to a unit vector. This detect element can therefore be used as a direction vector.

Examples:	Comments:
`circle 2`	Circle of radius 2
`circle -2 -3 -4`	Half-circles of radii 2, 3, and 4
`circle 10j10`	Circle arc extending 45 degrees
`circle 0j-1`	Unit three-quarter circle

`text` *point string* **`...`**

where the point is optional.

Presented as a paragraph of text with the top-left corner at the specified point, expressed in the local system (default is the origin). Each specified *string* forms one row in the paragraph. The text is parallel to the canvas.

On pointing, the (0-based) row and column indices of the detected character, counted from top left, are returned.

Examples:	Comments:
`text "Please state vicar's name here:"`	
`text "Amanita" "Mycena" "Lepiota"`	3-row paragraph
`text 5j5 "Amanita"`	Text shifted by <5, 5>

`menu` *point string* **`...`**

Behaves like `text` on presentation, except that if the first (or only) word in a row is the effective `state` name (see 3.3.5), the row is highlighted in a complementary background colour.

The main difference from `text` lies in the behaviour on interaction: on pointing at a `menu` the detected row is returned literally as a character string.

Examples:	Comments:
`menu "truite" "gigot" "fromage" "sorbet"`	Four-item menu
`menu “Extend forward”`	Highlighted in state `Extend`

Remarks: The interactive property of `menu` makes it useful for menu work. The Response program (cf. Section 2.6) may use the first (or only) word of the returned string as a keyword to fan out to detailed responses. Highlighting is useful to signal that a given menu choice is currently active. It is sufficient then to set the `state` parameter of the subpicture (sect. 3.3.5).

Implementation note: Through a DIPRO utility, the application may specify that a `menu` in a given subpicture be activated as a popup by right-button mouse clicks.

`array` *collist* **`/ ...`**

where *collist* is a list of *colcode*s.

Presented as an array of rectangular, coloured tiles, scaled to fill the local window. Each tile has the colour defined by the corresponding code. The code `none` causes the tile to be omitted; i.e. a hole appears in the array. The rows of the array are separated by a slash.

On pointing, the (0-based) column and row indices of the detected tile are returned, counted from the bottom-left corner.

Examples:	Comments:
`array blue white red`	French flag filling local window
`array blue white red / white red blue / red blue white`	*idem*, but chequered
`array none green / green green green / none green`	Green cross
`array 255 126 0 white blue / green`	Two-row array starting with orange

Remarks: An extension would allow the specification of a rectangle, onto which the array is mapped.

`image` *point string*

where *string* is the name of an image that was earlier loaded (with a DIPRO utility), and the point is optional.

Presents the image, centred at *point*, expressed in the local system (default is the origin). The image is parallel to the canvas.

On pointing, the (0-based) indices of the detected pixel are returned, counted from the bottom-left corner.

Examples:	Comments:
`image "score"`	Shows the image in the previously loaded file `score.bmp`
`image 7j6.5 "score"`	Image centered at <7, 6.5>

`locate` *grid*

where *grid* is a *point* whose components are truncated to integers.

Presented as (1) the rectangle that forms the local window frame, and (2) a coordinate grid in the frame. The *grid* value specifies the number of grid lines to be shown in the *x* and *y* directions. The grid is adjusted to pass through the local origin. If *grid* is specified as 0, only the frame is shown. If *grid* is negative, the primitive is not shown, and only used for pointing.

On pointing, the inside of the frame is detected, and its coordinate is returned, expressed in the local system.

Examples:	Comments:
`locate -1`	Invisible drawing area
`locate 5j4`	Both frame and grid shown (5 by 4 lines)

`locate`	`5`	*idem,* but no horizontal lines
`locate`	`1j1`	Frame with coordinate axes
`locate`	`0`	Only frame

Remarks: This primitive is useful to provide direct interaction with the pixels in a window, e.g. to let the user point out a location.

`keyin` *string mode*

where *mode* is optional.

Shown on presentation only if *mode* is the mnemonic `frame`. The local window frame is then outlined (as with `locate 0` above).

The primitive is used for text entry from the keyboard. The user interacts by pointing in the local window (cf. the `locate` primitive), and is then invited to enter the text in a small dialog, prompted by *string*. At normal closure of the dialog, the entered text is returned.

Examples:	Comments:
`keyin "Please state your password"`	Invisible pointing area
`keyin "why?" frame`	Window frame is shown

`key` *string*

where the characters in the string identify keys on the keyboard; system keys can not be specified.

Not shown on presentation.

On pointing (the user presses one of the keys in *string*), the corresponding key character is returned.

Examples:	Comments:
`key "Abc"`	Enable keys 'A', 'b', and 'c'
`key " "`	Enable space bar

`poster` *id name*

where *id* is a positive integer.

Not shown on presentation. This primitive interacts with the application through *posting*, rather than with the user.

On pointing (the application posts an event with the same *id*), *name* is returned.

Examples:	Comments:
`poster 21 beata`	Enable postings identified by 21; “beata” handles the response.

Remarks: The interactive property is similar to that of the `menu` primitive. The primitive is especially useful if *name* refers to a subpicture.

3.3.3 Geometric Parameters

Geometric parameters are described in terms of their general effect on the subpicture they are properties of, and in particular their effect on primitives. Their propagation is also defined (cf. Section 3.2.1). Each parameter is associated with a default value, which the Model propagates to the root subpicture when the Presentation or Correlation phase begins (Section 2.4).

`pos` *point*

Effect: The subpicture is shifted to the specified position, relative to its parent. Thus *point* becomes the local origin.

Default: `0` (i.e. no shift).

Affected primitives: All except `key` and `poster`.

Examples:	Comments:
`pos 4`	Shift right 4 units
`pos 4j-5.1`	Shift right 4 and down 5.1 units

`orient` *dir*

where *dir* is a non-zero *point* whose magnitude is ignored, or a real number followed by the letter `d` for degrees.

Effect: The subpicture is oriented (rotated) in the specified direction, relative to its parent. Thus the vector from the local origin to *dir* defines the local *x*-axis.

Default: `1` (i.e. no rotation).

Affected primitives: `figure, box, circle`; other primitives only as to their positioning. Thus texts, images, and arrays remain parallel to the canvas. However, the `text2` and `image2` primitives (Section 3.4.3) can be oriented.

Examples:	Comments:
`orient 4j4`	Turn 45° anti-clockwise
`orient 23j23`	*idem*
`orient 45d`	*idem*
`orient 1.732j1`	Turn 30° anti-clockwise
`orient 0j-1`	Turn 90° clockwise
`orient -1`	Turn 180°

Remarks: An extension might allow `array` to be rotated.

`scale` *factor*

where *factor* is a *point*. If the *y* component is omitted, the *x* component is assumed (isometric scaling).

Effect: The subpicture is scaled by the specified factor, relative to its parent. The two components of *factor* individually scale *x* and *y*. If a component is negative, the subpicture is mirrored in the opposite axis.

Default: `1j1` (i.e. no scaling).

Affected primitives: As for `orient`.

Examples:	Comments:
`scale 4`	Magnify isometrically 4 times
`scale 4j4`	*idem*
`scale 4j2.5`	Magnify anisometrically
`scale 0.5`	Reduce to half size
`scale 1j-1`	Mirror in *x*-axis
`scale -1j-1`	Mirror in both axes (i.e. mirror in the origin)
`scale 1j0`	Project onto *x*–axis (i.e. infinitely thin in *y*)

`skew` *dir*

where *dir* is a *point* `xjy`. Meaningful values for the components are in the range [-1,1].

Effect: The subpicture is skewed (distorted) by *dir*, relative to its parent. A positive *x* skews the subpicture to the right, negative to the left. Similarly, *y* skews the subpicture upwards or downwards.

Default: `0` (i.e. no skewing).

Affected primitives: As for `orient`.

Examples:	Comments:
`skew 0.4`	Skew right by 40%
`skew -0.4`	*idem,* but left
`skew -1`	Skew all the way to the left
`skew 0.5j0.5`	Skew both right and upwards

`window` *vertex1 vertex2*

where the values are *points*; *vertex2* is optional.

Effect: Clips the subpicture to the specified rectangular window frame. The frame is defined by the two vertices of either diagonal, expressed in the local system; if the second vertex is omitted, the negative of the first one is used (making the window symmetric around the origin). A window may be scaled, but only its centre is rotated by the `orient` parameter. The frame therefore always remains parallel to the canvas. The window frame is not explicitly displayed (cf. the `locate` primitive; Sect. 3.3.2). The parameter propagates under set intersection: the window propagated from a parent subpicture can not be expanded, only reduced.

Default: Infinity (i.e. no clipping).

Affected primitives: All; `key` and `poster` are handled as if positioned at the local origin.

Examples:		Comments:
`window`	`4j4 16j16`	Place window in 1st quadrant
`window`	`16j16 4j4`	*idem*
`window`	`4j16 16j4`	*idem*
`window`	`4j4`	Place window symmetrically around origin
`window`	`-4j-4 4j4`	*idem*
`window`	`0`	Make subpicture invisible

3.3.4 Cosmetic Parameters

Cosmetic parameters are described in the same terms as their geometric counterparts (Sect. 3.3.3).

`colour` *colcode mode*

where *mode* is an optional mnemonic.

Effect: The subpicture is presented with the colour defined by *colcode* (Sect. 3.3.1). If *mode* is omitted, the parameter is absolute. If *mode* is the mnemonic `blend`, the parameter is relative and propagates by blending. This means that the effective value of each colour component is the average of the specified and propagated values.

Default: `white`.

Affected primitives: All except `key, poster, array, image`.

Examples:	Comments:
`colour 255 255 0`	Set colour yellow
`colour yellow`	*idem*
`colour darkgreen blend`	Combine colours by blending

`style` *type*

where *type* is a mnemonic.

Effect: Lines in the subpicture are presented using the specified type. Certain line-types cause the vertices of figures to be unconnected and shown by marker symbols ("pin-point" figures). The parameter is absolute.

Supported line-types:

type	style
`none`	line not shown
`solid`	solid line
`bold`	thick, solid line
`dot`	dotted line
`dash`	dashed line
`dotdash`	dot-dashed line
`point`	vertices shown as points
`cross`	vertices shown as crosses
`diamond`	vertices shown as diamonds

Default: `solid`.

Affected primitives: `figure`, `circle`, and the grid in `locate`.

Examples:	Comments:
`style dot`	Set dotted lines
`style cross`	Set crosses as marker symbols
`style bold`	Set thick lines
`style none`	Set invisible lines

`font` *string size*

where the optional *size* is a positive integer.

Effect: Text in the subpicture is presented using the font named *string* and the given size in pixels. If *string* is empty, a "standard" font is assumed. The parameter is absolute.

Default: `"" 20`.

Affected primitives: `text`, `menu`.

Examples:	Comments:
`font "Arial" 14`	
`font "Courier New"`	Use default font size 20
`font ""`	Use standard font

`render` *density colcode*

where *density* is either an integer in the range [0, 100] or a mnemonic, and *colcode* is optional.

Effect: Sets the rendering pattern of the subpicture. If *density* is given as an integer, the interior of primitives in the subpicture are surface-rendered to the specified density, expressed in percent, providing more or less opaque surfaces. Zero density means no rendering, and 100 means solid rendering ("area fill"). If *density* is given as a mnemonic, the subpicture is rendered with a pattern (see below). The second value defines the colour of the rendering; if omitted, the effective colour is used. The parameter is absolute.

Default: `none`.

Affected primitives: `box`, `locate`, `figure`, and `circle`. If the primitive is open and rendered (even if by 0%), it is closed by the addition of a line segment.

Supported rendering patterns:

density	pattern
`none`	none
`solid`	solid
`cross`	cross-hatching
`vert`	vertical stripes
`horiz`	horizontal stripes

Examples:		Comments:
`render`	`100`	Turn on all pixels, surface opaque
`render`	`solid`	*idem*
`render`	`none`	No rendering
`render`	`0`	*idem*, in addition the primitive is closed if needed (see above)
`render`	`50`	Set half transparent
`render`	`50 pink`	*idem* with rendering colour
`render`	`cross pink`	Pink rendering with cross-hatches

3.3.5 Dynamic Parameters

Dynamic parameters are described in the same terms as the previous parameters. They are absolute and (except for `drag`) affect all primitives.

`state` *name*

Effect: The *state* of the subpicture is set to *name*, which must not be the name `off`. The parameter serves to control `vis` and `det` parameters in child subpictures as described below.

Apart from `on` and `off`, state names are invented by the programmer. They may coincide with subpicture names without conflict.

Default: `on`.

Examples:	Comments:
`state Adam`	Switch state to `Adam`
`state on`	Switch to the default state

`vis` *name* **`...`**

Effect: The subpicture is visible only if the specified list contains either the effective state name or the name `on`. If it does not, the subpicture is by-passed during both Presentation and Correlation phases.

Default: Empty name list.

Examples:	Comments:
`vis Adam`	Visible only in state `Adam`
`vis Adam Eve urgh`	Visible in either of these states
`vis on`	Visible in all states
`vis off`	Invisible

Remarks: If the `vis` parameter is specified, the subpicture may be switched on by using the default state `on`, and off by any state that is not used elsewhere (e.g. `off`).

det *name ...*

Effect: The subpicture is detectable through pointing only if visible, *and* if the specified list contains either the effective state name or the name `on`. If this is not the case, the subpicture is by-passed during the Correlation phase.

Default: Empty name list.

Examples:	Comments:
`det Eve`	Detectable only in state `Eve`
`det off`	Undetectable

drag *mode name*

where *mode* is a mnemonic (see below), and the name is optional.

Effect: Depending on *mode*, detectable primitives in the subpicture may be dragged with the mouse. Only primitives in the subpicture, not in its descendants are dragged. During dragging, the primitives are displayed without surface rendering.

The mode determines which geometric parameter in the subpicture is dragged:

mode	Dragging
`none`	nothing is dragged
`pos`	the position is dragged
`scale`	the scale is dragged
`xscale`	*idem*, but in *x* only
`yscale`	*idem*, but in *y* only
`orient`	the orientation is dragged

The user drags a subpicture by pointing at it with the left mouse-button, then moving the mouse to a new location before releasing the button. This event contributes two detect elements to the Response Queue (see the introduction to 3.3.2):

1. The *name* (default is the name `drag`).
2. The change in the geometric parameter indicated by *mode*, expressed in the local system:
   - `pos`: the vectorial displacement.
   - `scale`, `xscale`, or `yscale`: the rectangular scale change.
   - `orient`: the angular change, expressed as a direction vector.

Affected primitives: `circle`, `figure`, `box`.

Default: `none`.

Examples:	Comments:
`drag pos mydrag`	Position dragged and `mydrag` handles the result
`drag xscale`	Scale dragged in *x*-direction and `drag` handles the result

Remarks: DIPRO does not update the geometric parameter of the subpicture at the end of dragging. The response associated with *name* may, however, achieve this by combining the second detect element with the local parameter (cf. Section 2.6).

3.3.6 Links

link *name ...*

Effect: The subpicture is linked to the subpictures named in the list, making the latter children of the former. An undefined subpicture in the *name* list is linked but treated as empty.

Examples:	Comments:
`link urgh`	Link to subpicture `urgh`
`link Oscar Beata`	Link to two subpictures

`name` *name*

Effect: The subpicture is renamed to *name*, provided the name is free. All links to the subpicture are updated to reflect the name change.

Examples:	Comments:
`name Eve`	Rename subpicture to `Eve`

3.3.7 Additional Operations

The syntax is extended beyond Section 3.3.1 by a few additional operations on subpictures. The statement syntax then becomes:

name oper, ...;

where *oper*, beside being a *property*, may be either of:

`unlink` *name1*	Remove a link to subpicture *name1,* if any, in subpicture *name*
`clear`	Remove all properties (except `name`) from subpicture *name*
`print`	Print all properties of subpicture *name* to a log file
`copy` *name1 typename*	Copy property *typename* from subpicture *name1* to subpicture *name*
`copy` *name1*	Copy all properties (except `name`) from subpicture *name1* to subpicture *name*

Examples:	Comments:
`oscar unlink Beata;`	Remove the link to `Beata` from subpicture `oscar`
`oscar print, clear;`	Print all properties of subpicture `oscar`, then remove them
`oscar copy Beata;`	Copy the properties of subpicture `Beata` into `oscar`

3.4 An Experimental 3D Subset

The following set of three-dimensional primitives and parameters has been used in DIPRO implementations to demonstrate various 3D techniques (wire-frame modelling, generalised polytopes, spline curves, 3D rotation, perspective, stereo). Basically wire-frame objects are provided, but there is limited support for hidden-surface cueing.

The device coordinate system is a proper extension of the 2D system described in the previous section. As you look at the canvas on the screen, by default the *x*-axis is towards the right, the *y*-axis upwards, and the *z*-axis towards you. The canvas forms the *xy*-plane. But the 3D position parameter will shift the coordinate system, and the 3D orientation parameter will turn it in any other direction.

Most 2D parameters and primitives are extended to 3D by the addition of a *z* component to each coordinate, using a pseudo-quaternion form. Thus the notation:

```
2j3k4
```

represents the 3D coordinate <*x,y,z*> = <2,3,4>. Unless stated otherwise, the 2D form `2j3` is handled as an abbreviation of `2j3k0`, and therefore specifies a coordinate in the local *xy*-plane. The syntactic token *point* now denotes either, and lists of *point*s may contain any mix of 2D and 3D points.

On presentation and correlation, all primitives are projected onto the canvas (optionally under perspective), and clipped by the effective 2D window.

3.4.1 Extensions to 2D Primitives

`figure` *pointlist / ...*

The 3D form is upwards compatible with the 2D form, differing only in that points may be 3D.

Examples:	Comments:
`figure 3j4k5 -3j4k5 7j9`	3-vertex 3D polygon
`figure 3j4k5`	Single 3D dot
`figure 0j0k5 -3j4k5 7j9k5 0j0k5`	Triangle at height 5
`figure 0 1 / 0 0j1`	The *xy* coordinate axes
`figure 0 1 / 0 0j1 / 0 0j0k1`	The three coordinate axes

`box` *point ...*

The 3D form is upwards compatible with the 2D form, and differs only in that boxes may be 3D. Rendering and interaction follow the same rules as described for the new 3D primitives (Section 3.4.2). If the *z*-component of a *point* is omitted, the *y*-component is assumed.

Examples:	Comments:
`box 3j4k5`	Box with a vertex in <3, 4, 5>
`box 5j5k5`	Cube of size 10
`box 5j5`	*idem*
`box 5`	*idem*
`box 3j4k0`	Flat box (rectangle) in *xy*–plane
`box 0j4k3`	*idem* but in *yz*–plane
`box 3j4k6 6j8k12`	Two concentric boxes
`box 3j4k0 6j8k12`	One 2D and one 3D box

`circle` *point ...*

The 3D form is upwards compatible with the 2D form, and differs only in that the ending *point* may specify a *z*-component. If this component is non-zero, the primitive presents a helix or a helical arc. Helices may be tiled (see Sect. 3.4.5).

Examples:	Comments:
`circle 2j0k5`	Helix of radius 2 and height 5 (sine curve, if presented with `orient 0j1 0j1`)
`circle 2j0k5 4j0k-5`	Helices of radii 2 and 4
`circle 10j10k5`	Helical arc extending 45 degrees

`text` *point string ...*
`menu` *point string ...*
`image` *point string*

The 3D forms differ from the 2D forms only in a possible 3D position.

Examples:	Comments:
`text 3j4k5 "Urgh"`	One-row text with 'U' at <3, 4, 5>
`text 3j4 "Urgh" "Blurgh"`	Two rows at <3, 4, 0>

3.4.2 3D Primitives

The following family of three primitives provides variations of these familiar objects: prism, cylinder, pyramid, cone, octahedron, and sphere.

General specifications:

1. The primitives are all presented as a set of regular *n*-gonal faces, parallel to the local *xy*-plane, centred on the local *z*-axis, and symmetrically placed around the local origin. The faces are oriented so that one vertex projects onto the positive *x*-axis. By default, the primitive has two faces (if a pyramid, one reduces to the apex), but you may interpolate *m* additional faces. The non-negative integers *n* and *m* are given as optional values.

2. The mandatory value *size* specifies the size of the object. It is a *point, r*`j`*z*, where *r* is the radius of a circle in which one of the faces, called the *base face*, is inscribed, and where *z* is the half-height.

3. Longitudinal edges, connecting the face vertices, are shown but not detectable.

4. Rendering affects only the faces of the object, and line style only the longitudinal edges.

5. The user may interact by pointing inside one or more of the faces. On pointing, a list of 0-based indices is returned. Each index identifies a detected face, counting from the bottom (lowest *z* in the local system).

`prism` *size n m*

Presented as a straight prism. The two end faces have one vertex at <*r*`,0,`*z*> and <*r*`,0,-`*z*>, respectively.

m equally spaced faces are inserted between the end faces of the prism. If you omit *n* and *m*, `3 0` are assumed, i.e. the prism is triangular.

Examples:	Comments:
`prism 10j8 5`	Pentagonal prism (height 16)
`prism 10j-8 5`	*idem*
`prism 10j8 5 4`	*idem*, but with 4 additional faces
`prism 10j8`	Triangular prism
`prism 10j7.07 4`	Cube
`prism 10j8 33`	Cylinder (almost)
`prism 10j8 2`	Rectangular plane, confluent with *z*-axis
`prism 10j8 2 4`	*idem*, but striped
`prism 10j0 7`	Heptagon, confluent with *xy*-plane

`pyram` *size n m*

Presented as a straight pyramid. The base face of the pyramid has one vertex at <*r*`,0,-`*z*>, while the apex is located at <`0,0,`*z*>.

m equally spaced faces are inserted between the base and the apex of the pyramid. If you omit *n* and *m*, `3 0` are assumed, i.e. a tetrahedron is presented.

Examples:	Comments:
`pyram 10j8 5`	Pentagonal pyramid (height 16)
`pyram 10j-8 5`	*idem*, but pointing downwards
`pyram 10j8 5 3`	*idem*, but with 3 additional faces
`pyram 10j15 33`	Cone (almost)
`pyram 10j7.07`	Regular tetrahedron
`pyram 10j10 2`	Triangle, confluent with *z*-axis

basket *size n m*

Presented as a "basket", resembling a wire-frame globe of radius *r* and half-height *z*. The basket consists of *m* faces, the largest of which has one vertex at <*r*,0,0> if *m* is odd. The corresponding vertex of the other faces is spaced between the poles of the globe, at equal distances along the perimeter of an ellipse with half-axes *r* and *z*.

If you omit *n* and *m*, `4 1` are assumed, i.e. an octahedron is presented.

Examples:		Comments:
`basket`	`10j10 9 5`	Basket of 5 nonagons and radius 10
`basket`	`10j10 25 19`	Sphere (almost)
`basket`	`10j5 25 19`	Ellipsoid (almost)
`basket`	`10j0 25 1`	Flat wheel (almost)
`basket`	`10j2 25 2`	Somewhat thicker wheel
`basket`	`10j10`	Regular octahedron
`basket`	`10j10 3`	Tetrahedral die
`basket`	`10j15 25`	Top (almost double cone)

spline *size tan1 tan2*

where *size* is a real number and the last two values are optional 3D points.

Presented as a 3D cubic spline curve whose base-line lies on the local *x*–axis, centred at the origin. The beginning point of the spline is at <-*size*, 0, 0> and the end point at <*size*, 0, 0>. The tangent at the beginning point is given by *tan1*, at the end point by *tan2*. If *tan2* is omitted, *tan1* with a negated *y*-component is used, making the spline symmetric around the *y*-axis. If also *tan1* is omitted, its default is `1j1`.

The spline is affected by the `render` parameter, which also closes the primitive by showing the base-line. The primitive is not affected by the `style` parameter, but it may be tiled (see the `tile` parameter, sect. 3.4.5).

On pointing, the perimeter of the spline is detected, and the tangent at the detected point is returned as a 3D vector.

Examples:	Comments:
`spline 20 2j4 2j-4`	2D symmetric spline of half-length 20
`spline 20 2j4`	*idem*
`spline 20 2j4k-1 3j4k4`	3D spline (to be viewed with 3D orientation)
`spline 20 4.5j6 5.5j-2`	Spline with a loop
`spline 20 2.5j5 1.5j6`	Spline crossing base line

3.4.3 Transformable text and image

The following two primitives are similar to `text` and `image` (Section 3.3.2), but are affected by all geometric parameters, including 3D (performance penalty). Also their behaviour on interaction is analogous.

text2 *point string ...*

where the optional 2D *point* is the vertex of a rectangle onto which the text is mapped, expressed in the local system. The rectangle is centred at the local origin and parallel to the local axes.

Presented as the `text` primitive, but as the mapping is expressed in the local system, the text is transformed by geometric parameters. However, if *point* is omitted, the text extent is used for mapping, which means that the primitive becomes insensitive to the `scale` parameter.

If the *y* component of *point* is omitted, the *x*-component is assumed. If *point* specifies a vertex other than the top-right one, the text is mirrored accordingly.

Examples:	Comments:
`text2 40j3 "Please state the name:"`	Text mapped onto rectangle through <40,3>
`text2 -10j3 "Please"`	Backwards text
`text2 "Adam" "" "Eve"`	Paragraph mapped onto own extent

`image2` *point string*

For mapping details this primitive is analogous to the `text2` primitive (above). On presentation and interaction it behaves as `image`.

Examples:	Comments:
`image2 10j10 "score"`	Image mapped onto square through <10,10> (size 20)
`image2 10 "score"`	*idem*
`image2 "score"`	Image mapped onto own extent
`image2 10j-5 "score"`	Image mapped and shown upside-down

3.4.4 Extensions to 2D Parameters

These are all upwards compatible with the 2D form. The parameters affect primitives as described in Section 3.3, as well as all described in Sections 3.4.2-3.4.3.

`pos` *point*

Effect: The subpicture is shifted to the specified position, relative to its parent. Thus *point* becomes the local 3D origin.

Examples:	Comments:
`pos 4j5k6`	Shift by <4, 5, 6>
`pos 4j5`	2D shift
`pos 0j0k-18.2`	Move backwards 18.2 units

`scale` *factor*

where *factor* is a 3D *point*. If the *z* component is omitted, the *y* component is assumed.

Effect: The subpicture is scaled individually in *x*, *y*, and *z* by *factor*, relative to its parent. If a component of *factor* is negative, the subpicture is mirrored in the opposite plane.

Examples:	Comments:
`scale 2`	Double the size
`scale 2j2k2`	*idem*
`scale 2j2`	*idem*
`scale 0.5j0.5k1`	Reduce by half in *xy*-plane
`scale 1j1k0`	Project onto the *xy*-plane (i.e. flatten)
`scale 1j1k-1`	Mirror in *xy*-plane

`orient` *dir axis*

where *dir* is a 2D *point*, and *axis* is an optional 3D *point.* The magnitudes of both values are ignored and both must be non-zero; *dir* may also be given in degrees (see Section 3.3.3).

Effect: The subpicture is oriented (rotated) in the direction given by *dir,* relative to its parent. It is rotated around an axis extending from the local origin to the point *axis*. If *axis* is omitted, `0j0k1` (the *z*-axis) is assumed, and the parameter reduces to its 2D form.

Examples:	Comments:
`orient 1.732j1 1`	Pivot 30° around *x*-axis
`orient 30d 1`	*idem*
`orient 10j17`	Pivot around *z*-axis
`orient 10j17 0j1k1`	Pivot around axis in *yz*-plane
`orient 10j-17 0j1k1`	*idem*, but pivot in opposite direction
`orient 10j17 0j-1k-1`	Same effect
`orient 0j1 1`	Provide side view
`orient 0j1 0j1`	Other side view

`skew` *dir1 dir2*

where *dir1* and the optional *dir2* are 2D *point*s, *x*`j`*y*. Meaningful values for the components are in the range [-1,1].

Effect: The subpicture is skewed (distorted) in *x* and *y* by the given factors, relative to its parent. A positive *x* skews the subpicture to the right, negative to the left. Similarly, *y* skews upwards or downwards. Skewing by *dir1* depends on the shape in *xy*, and skewing by *dir2* depends only on *z*. Thus, the second value can be used with the 3D primitives to produce oblique prisms, pyramids, cones, etc.

If *dir2* is omitted, 0 is assumed (i.e. no *z*-dependent skewing), and the parameter reduces to its 2D form.

Examples:	Comments:
`skew 0 0.5`	Skew towards *x* in *z* (for oblique prism, etc.)
`skew 0 0.5j0.5`	Skew both right and upwards

3.4.5 3D Parameters

`persp` *dist sep*

where *dist* and the optional *sep* are real numbers.

Effect: This parameter provides perspective. On presentation of the subpicture the user is assumed to view the centre of the canvas at the distance *dist*. The values are expressed in the canvas system (root subpicture; see Section 3.2.3). A negative *dist* gives the illusion of the object being a hole in a solid background.

If *sep* is non-zero, it is taken as half the user's eye separation, and two perspective projections are shown, providing a stereo view. Your eyes are assumed to be at positions <*sep*, `0`, *dist*> and <`-`*sep*, `0`, *dist*>, respectively, relative to the centre of the canvas. You may permute the stereo-pair with a negative *sep*. The parameter is absolute.

Default: Infinity and `0` (i.e. no perspective and no stereo).

Affected primitives: all that use 3D. Points in the effective *xy*-plane are unaffected.

Examples:	Comments:
`persp 50`	Perspective at 50 units from screen, see below
`persp 50 5.3`	*idem* with stereo
`persp 5`	Frog perspective
`persp -50`	Object appearing as a hole
`persp 9999`	No perspective (almost)

Suppose the root specified `window 25j25`, leading to a canvas width of 50 units. Then the first two examples assume a viewing distance equal to the canvas width.

`tile` *width*

where *width* is a real number.

Effect: This parameter adds a tiled surface to the primitive so as to emphasize the spatial dimension. The surface subtended by the primitive and a base-curve in the *xy*-plane is split into tiles of the given width, expressed in the local system. The base-curve depends on the primitive (see below). If *width* is `0`, the primitive is not tiled. Tiling is shown as longitudinal edges, affected by the `style` parameter. The parameter is absolute.

Default: `0` (i.e. no tiling).

Affected primitives:
`circle` and `figure`: the base-curve is a projection of the primitive onto the *xy*-plane.
`spline`: the base-line is used as the base-curve.

Examples:	Comments:
`tile 2.3`	Add 2.3 wide tiles
`tile 0`	No tiling

4. Application Building

Several analogies, as well as differences, between subprograms and subpictures were pointed out in the introductory sections. In particular (Section 2.6), there is often a correspondence between the two types of constructs in a well-structured application. This works in such a way that the code to be executed in response to the user pointing at a subpicture `S` can often be gathered in a function `R`, which then becomes a polymorphic method of the subpicture class. Such a function `R` will be called a *response method*. This object-response connection forms the basis for a host of advanced application-building techniques.

It is, for example, highly productive for the designer to be able to specify the canonical form of `S` together with the algorithmic code `R`. A sufficiently smart program editor may provide separate windows on the screen for the two parts. Even higher productivity is reached if the editor is able to display the graphic representation of `S` as it is developed and debugged. On saving (and possibly compilation), the parts are automatically sent to the Dialog and Problem Processors respectively. An example:

Subpicture canonical form `S`

```
oscar
   colour blue,
   text "No Title",
   pos 3j5;
```

Response method `R`

```
oscar::response() {
 newtitle = readrec("bookfile.txt", RQ);
 // update subpic
 oscar#text = newtitle;
}
```

Not all functions in an application will be response methods (an example could be the calculation of a mean value); nor will all subpictures have a response. But on the whole it is possible, and indeed profitable, to develop an application in such a way that all first-level program code is made up from response methods. Remaining code is then invoked by the responses as second-level functions (`readrec` in the example). Even the main function may be regarded as the response to a *latent event*, which is automatically posted to the Dialog Processor at application start-up.

Section 2.5 emphasises the crucial role of the Response Queue for directing the flow of control from one response method `R1` to another `R2`. The polymorphic invocation of the methods is typically achieved by a dispatching (fan-out) utility function based on the queue elements. When invoked, each `R1` needs to know at most the identity of the following `R2` to be invoked. This information is provided by a built-in system variable (`RQ` in the example; cf. Sect. 2.6), which, at the end of the Response Queue, also yields the pointing data associated with the detected primitive instance (value of the `text` primitive in the example).

Another important technique, mentioned earlier, is *state networking*. It simplifies a complex dialog with many interactive choices, which is why the example set (Section 3.3.5) incorporates *dynamic* parameters specifically designed to assist in this area. Different states regulate different paths through the picture structure, either in parallel (complementary picture parts), or in series (mutually exclusive picture parts). The dialog states are also available (through the reference constructs suggested in Section 2.3) to the response methods, where they may serve to regulate the logic.

With some experience, a designer using DIPRO together with a sophisticated editor can often build the rudiments of a graphics application in a matter of minutes. The resulting code is immediately available for quick on-line testing.

In summary, DIPRO provides a natural and comprehensive way to structure an interactive application program, with a considerable boost in design productivity. This has been shown by many small to medium-sized applications developed by the author and participants in his classes.

5. The Theory of DIPRO

This section describes the *Dialog Processor Model* in a formal way, using familiar concepts from set theory and object-oriented design[5].

Central to the Model description is the notion of a *picture*. Ultimately a picture is something that can be perceived by the human eye. But before a machine shows a picture to a human, it may store the information representing the picture in different formats, depending on the level of abstraction. These representations will here be called *picture spaces*.

Just as a program is made up of a number of subprograms or functions that are linked by calls, it is useful to treat a picture as made up of *subpictures*, linked by relationships, such as "consists-of" (and its inverse "contained-in"). The analogy, often evoked in graphics literature, stumbles when one compares some details of the program or picture objects, but at a conceptual level it provides a symmetry that is, in fact, more than accidental (Section 5.2.2).

Any theory that deals with structured pictures should state what a subpicture consists of, how it is identified and referred to, how the structure is built, and the role of the subpicture in the total picture. There should also be a discussion on how transformations and other parameters propagate, which in the present theory is based on the notion of "effective value" (Section 5.1.3). Where most other models are lacking, however, is in the very important area of event/response formalism with feedback from response to picture (Section 5.2). This closes the loop presentation—interaction—response (Section 5.3).

Finally, a model may also state how it accommodates the related Application Programming Interface (API), in particular how a subpicture is queried and updated. Here a mechanism based on object properties is proposed (Section 5.1.5 and cf. Soop 1988), since this can easily be adapted to a variety of API protocols.

5.1 Picture Structure

5.1.1 Subpictures

Definition: A ***subpicture***[6] is a named object, characterised by a number of ***properties***, which fall into the following broad classes:

- ***Primitives***
- ***Parameters***
- ***Links*** to other subpictures

Example: This example (expressed in some imaginary language) specifies the properties of a subpicture `Oscar`:

```
subpic Oscar
   colour red,                 // colour is a parameter
   polyline (2 3) (0 10),      // primitive
   scale 2 3,                  // parameter
   link Beata Fred,            // Beata and Fred are other subpix
   text (0 5) "URGH";          // primitive
```

Discussion:

1. Apart from *links*, the Model does not prescribe specific subpicture properties. These must be formally specified by any implementation of the Model, e.g. by an international standard.

---

[5]As is customary, the symbols "`: , | ⊂ ~`" and "`{}`"stand for, respectively, "then", "and", "such that", "included in", "not", and "set of", while ∀ means "for all" and ∃ means "there exists".

[6]In the formal prose, bold-italics are used to define or introduce new terms, and italics refer to terms previously defined or assumed to be known.

2. The term *parameter* is here used to embrace the notion of an attribute, e.g. colour or visibility, as well as geometric transformations. As is well-known from structured graphics systems, these tend to have similar behaviour.

3. Although the Model does not prescribe specific *parameter* properties, it would be difficult to imagine an implementation without a `window` parameter. As discussed in Section 3.2.3, it can be defined to include the properties of both a clipping window and a viewport.

4. The notion of a *link* property is used to span the picture space with a hierarchical structure. This is preferred to saying that a subpicture "contains" another subpicture, since linking underlines some important mechanisms, common in extant graphics systems: (1) a subpicture `S` can be included as several instances in the total picture, and (2) modifying `S` then has a potential effect on all its instances. This means that a *link* forms what is sometimes formally termed an *instance connection* between objects. Note also that essentially "flat" implementations (such as GKS), are accommodated simply by restricting the number of levels formed by the links.

5. Subpictures are similar to Minsky *frames* [Minsky 1975], where the main difference is that *links* are properties rather than consequences of subpicture creation. The reason for this is greater homogeneity in the Model; for example, the program can change the structure simply by updating a *link* property, in the same way as it changes the colour of a subpicture from red to green.

Rule: A subpicture *property* is defined by its ***type*** and its ***value***. The *type* is a member of an implementation-defined set. The fact that a subpicture `S` has a property of type `T` and value `V` is stated as an ordered triplet:

$$\langle S,T,V \rangle$$

Rule: The domain of *property values* depends on the *type*, but always includes a ***null*** value. Composite values (e.g. arrays) are allowed. In the following the expression $\langle S,T,null \rangle$ will be used interchangeably with the expression "property `T` is missing in `S`".

Rule: A subpicture `S` has at most one *property* of any given *type*:

$$\langle S,T,V \rangle, \langle S,T,W \rangle : V=W$$

Definition: A ***picture space*** `P` is a (possibly empty) set of triplets:

$$P = \{\langle S_i,T_i,V_i \rangle\}, \; i=1...n, \; n \geq 0$$

where the triplets are defined as above.

Definition: A *subpicture* `S` is said to be ***in*** the *picture space* `P`, if it has at least one property in `P`:

$$In(S): \exists T,V \neq null \mid \langle S,T,V \rangle \subset P$$

In the following, subpictures and their properties are implicitly discussed in the context of a given *picture space*. For brevity the phrase "$\subset$`P`" is then omitted for triplets.

Discussion: In the following, definitions and rules for naming, storing, and loading *picture spaces* are omitted for brevity. It is assumed that an application uses at most one *picture space* at a time.

Rule: An implementation of the Model shall provide a *link* property, here called *`link`*, whose *value* is a (possibly empty) set of subpicture names.

Discussion: The phrase "subpicture name" is here and in the following used loosely to designate any implementation-defined way of referring to a subpicture. It may involve character strings, references, indices, or pointers, just to list a few.

Definition: If a subpicture `S` has a *link* to another subpicture `Z`, `S` is said to be a ***parent*** of `Z`, and `Z` is said to be a ***child*** of `S`. This is expressed by the predicates:

Parent(S,Z): ∃M | <S,*link*,M>, Z⊂M
Child(Z,S): *Parent*(S,Z)

for some set M. By associativity, one may also speak about ***ancestors*** and ***descendants***, through any number of linkage levels.

Rule: A subpicture may have zero or more *children*, and zero or more *parents*.

Corollary: Since the value of the link property is a set, all *children* of a subpicture must be distinct.

Definition: A ***root*** subpicture R is one without a *parent*:

Root(R): ∀S, ~*Parent*(S,R)

An implementation of the Model may restrict *links* to be present only in *root* subpictures in order to enforce an essentially flat *picture space*.

Rule: No subpicture may be its own ancestor (or descendant). In other words, the picture space is spanned by a re-entrant tree (or acyclic, directed graph) through the *links*.

Discussion:

1. One might argue that a tree is too restrictive, and that a formal model should allow more general structures, such as those provided by the Entity-Relationship theory. But extant systems used in production graphics are almost invariably either flat or hierarchical. In my opinion, generalising into arbitrary linkages would risk making the Model virtually useless by omitting some very powerful mechanisms.

2. It is tempting to define a *picture* as a subpicture without a parent (including by associativity all primitive instances that can be reached from the root), and similarly a *primitive* as a subpicture without a child. None of this is needed for the theory, however.

   Moreover, one may note that the latter point is not wholly consistent. If one wants to integrate primitives into the *picture space*, they should not be defined as subpictures without a structure, but as subpictures whose inner structure is given by the graphics system, and cannot be manipulated by the programmer (the *mini-structure* of Soop [1982]). This is analogous to primitive functions (e.g. "+") in programming. A primitive property can then be regarded as a *link* to the primitive subpicture, and the primitives will form the true leaves of the picture space (cf. Fig. 2).

3. A *picture space* may contain more than one *root* subpicture, since there may be fragments of trees lying around, which are temporarily unlinked and ignored by the Dialog Processor. Also, the application may maintain a number of trees (*forest*), from which it selects one to present a picture.

4. This Model only provides one class of objects that can be named and referenced, namely *subpictures*. *Links* cannot be named, nor can individual *primitives* or *parameters* in a subpicture. Note, however, that the phrase "*subpicture* S" in the formalism denotes either a reference to the object or to its name S, depending on the context. The Model does not specify how the association between a subpicture and its name is achieved; one possible such mechanism might be a specific name parameter (cf. Sections 3.3.5, 6.3.2).

5. Being an object property ("has-a" relationship) is straightforward for a *parameter* (such as colour), but perhaps less obvious for a *primitive*. But one can argue that a primitive, say polyline, expresses a dual property of its subpicture, namely to (a) display a set of straight lines, (b) allow the user to interact by pointing at these lines.

Definition: A ***path*** H is an (ordered) list of at least one subpicture name, such that each name (except the last) denotes a parent of the next:

H = <S_1>, or
H = <S_1, S_2, ..., S_n> | *Parent*(S_i,S_{i+1}), i<n, n>1

Definition: Each distinct *path*, whose first and last elements denote two subpictures `R,S`, represents one ***instance*** of `S` with respect to `R`. If `S` has a primitive property of type `T`, the path also represents an *instance* of the *primitive* `T` with respect to `R`. This may be expressed by the predicate *`Inst`*:

$$Inst(S,R) : \exists H \mid S_1 = R,\ S_n = S$$
$$Inst(T,R) : \exists S,V \neq null \mid Inst(S,R),\ \langle S,T,V \rangle$$

where `H` is defined as above.

5.1.2 Primitive Properties

Definition: Only *primitives* have a direct visual effect on presentation, in the sense that a subpicture without a primitive, either in itself or in any of its descendants, can never be visible.

Discussion: The reverse does not hold, of course, in practical cases. A subpicture with primitive properties need not be visible; for example, its `visibility` parameter may be `off`, the primitives may be clipped away by the `window` parameter, the `colour` may be the background, or the `linewidth` zero.

Rule: A *primitive property* is characterised by the following sub-properties:

1. Domain of its *value*
2. Geometric shape
3. Interactive properties

An implementation of the Model shall specify these for all its primitive *types*.

Discussion:

1. As regards sub-property 1, API details are, of course, not part of the Model, so the exact syntax for specifying a value can not be prescribed. What an implementation must state is the domain and general format of the corresponding value at the API level. For example, the value of a `text` primitive might be specified as follows: "<*coordinate*><*char-string*>, where the coordinate is optional".

   Note that an implementation may allow composite values in primitives of any type `T`. This is the way to specify several `T` primitives (e.g. three `polylines`) in one subpicture.

2. As regards sub-property 2, it is a well-known fact that primitives can mimic each other. For example, a `circle` primitive can be mimicked by a `polyline`, or by the character "O". Naturally there is a many-to-many correspondence between primitive types and shapes, and only an indication of the intended shape is required in implementations. Thus, a `circle` primitive should produce something that looks like a fair approximation of a circle. A precise definition of what is meant by "fair" is meaningless. As is customary in standards work, the Model leaves it up to "marketing forces" to decide whether someone wants to release an implementation where this primitive looks like a square. — Also note that some primitives may have no geometric shape at all, i.e. nothing is presented.

3. Sub-property 3 should specify how one can detect the primitive through *pointing* (Section 5.2.1). It includes criteria like "hit windows" and tolerances, and whether, for example, the interior of closed shape is excluded from detection.

5.1.3 Parameter Properties

Definition: Parameters affect the visual or interactive behaviour of the subpicture. A parameter is either of:

Geometric: it affects the *spatial* appearance of the subpicture.
Cosmetic: it affects the *style* of the subpicture.
Dynamic: it affects the *state* of the subpicture.

The effect of a given parameter may vary with the primitive it is applied to and may even be absent for some combinations.

Discussion: This taxonomy of parameters is perhaps debatable, and may in fact be superfluous in the Model. The intent is merely to facilitate grouping in graphics-system documentation. It is easy to see that colour, line type, and hidden-surface cueing are cosmetic in nature, and that rotation and perspective are geometric. But a parameter like depth cueing, affecting, as it may, the geometry of the primitives could be a borderline case. An example of a dynamic parameter is `detectability`.

Definition: With reference to a subpicture `S` and a *parameter* of *type* `T`, such that `<S,T,V>:`

- `V` is said to be the ***local*** value of `T` in `S.`
- The value actually used in presenting a given *instance* `I` of `S` (i.e. having a direct effect on its primitives) is said to be the ***effective*** value `E` of `T` in `S`. This is written *`Eff`*`(S,I,T,E)`.
- The *effective* value of `T` in the *parent* subpicture of `S` in `I` is called the ***inherited*** value of `T` in `S.` If `S` has no parent in `I`, the inherited value is the ***default*** value of `T`. The *default* is an implementation-defined sub-property of `T` (see below).

Rule: All parameters ***propagate*** as follows. Consider an *instance* `I` of subpicture `S` and a *parameter* of *type* `T`, such that `<S,T,V>`:

Subrule 1: If the *local* value `V` is *`null`*, the *effective* value `E` of `T` is the *inherited* value.

Subrule 2: If the *local* value `V` is not *`null`*, the *effective* value `E` of `T` is either of the following:

- ***Relative*** parameter `T`: `E` is `V` ***combined*** with the *inherited* value (see below).
- ***Absolute*** parameter `T`: `E` is `V`. (This may be handled as a special case of a *relative* parameter.)

Subrule 3 : The *default* value of a *relative* parameter must be chosen as the *identity element* under combination.

Let `H` be the *path* defining the instance `I` (Section 5.1.1). Subrules 1-2 can then be summarised by using the predicate *`Eff`*:

`Eff``(S,I,T,`*`c`*`(e,V)): ∃Z⊂H | `*`Parent`*`(Z,S),` *`Eff`*`(Z,I,T,e)`
`Root``(S):` *`Eff`*`(S,I,T,D)`

where `D` is the *default* value of `T`, and *`c`* is the combination algorithm, such that:

`c``(e,`*`null`*`) = e`

Subrule 3 can be summarised thus:

`c``(D,V) = V`

Rule: A *parameter property* is characterised by the following sub-properties:

1. Domain of its *value*
2. Propagative sub-properties:
   - *relative* or *absolute*
   - *default* value
3. Effect on all *primitive* types

An implementation of the Model shall specify these for all its parameter *types*.

Example: As regards sub-property 3, an implementation of the Model might specify that circles are unaffected by rectangular scaling (which would turn them into ellipses). A `cellarray` primitive may be insensitive to a `colour` parameter, if it contains its own colour information.

Rule: An implementation of the Model shall specify a ***combination algorithm*** (Subrule 2 above) for each *relative* parameter `T`. Apart from the *local* and *inherited* values of `T`, this algorithm may depend also on the *inherited* or *effective* values of parameters of other types, but not on their *local* values.

Subrule: As this rule may engender infinite regression among the combination algorithms[7], any implementation must prescribe their exact dependence and order of evaluation (Section 5.1.5).

5.1.4 Notes on Subpicture Order

The definition of a subpicture as an object vested with properties leads to an explicit lack of *specification order*. The following three program sequences, expressed in the syntax of Section 3.3.1 and setting some properties of a subpicture `beata`, would therefore be equivalent:

```
beata colour blue, text "abc";
beata text "abc", colour blue;
beata text "abc"; /*......*/ beata colour blue;
```

As far as *parameter* properties (such as `colour`) go, it is easy to look at these as applicable to the subpicture as a whole, and their place in the subpicture object should therefore be unimportant. The situation for *primitive* and *link* properties is somewhat different, since their "order" may be revealed by over-painting when the subpicture is presented. But if this effect is important, then the implementation should provide a `priority` parameter, or use 3D, to sort out overlapping instances.

This is in sharp contrast to systems like PHIGS [1985], which has a complicated scheme involving line numbers in structures. Such schemes, designed to facilitate updating, obscure the fundamental object properties and complicate the API.

Apart from the specification order, there is also a processing order which is important in the definition of the Dialog Processor (see the next section).

5.1.5 Methods

Definition: The following five methods are defined for *subpicture* objects. They are taxonomically divided into *editing*, *tracing*, and *responding* methods:

Editing	***Tracing***	***Responding***
query method	*present* method	
set method	*correlate* method	*response* method

Rule: An implementation of the Model shall provide an API for the *editing* methods, where the implicit argument is a *subpicture* `S`:

***query*:** Formal argument: *property-type*
Returns the (*local*) *value* of the requested *property* of `S`.

***set*:** Formal arguments: *property-type, value*
Replaces the (*local*) *value* of the requested *property* of `S`.

The value returned by *query* or given to *set* may be `null`.

Definition: An ***empty subpicture*** is one for which the *query* method yields the `null` value for all *types*.

Discussion:

1. The *editing* methods can be described by regarding the *picture space* as an *associative memory* containing a set of `<S,T,V>` triplets (cf. Section 6). This emphasises the definition of an empty subpicture. In such an implementation, subpictures need not have an existence property: any bona-fide

[7]Example: c_1`(e1,v1)`, where `e1` depends on c_2`(e2,v2)`, whose `e2` depends on c_1.

literal S is the name of a subpicture in the *picture space*, which can be non-empty only if it has executed the *set* method with a non-*null* value.

2. Note, that for editing purposes the subpicture name or reference constitutes the only mechanism for addressing information in the *picture space*. If, in line with some systems, an implementation chooses to use "tags" to address information inside a subpicture, it must add a new link level, on which tags are defined as subpicture names in a reserved (possibly numeric) domain.

Definition: A *tracing* method produces all *primitive instances* (cf. 5.1.1) with respect to a *subpicture* S, given as the implicit argument. For each *primitive instance* I, it produces:

1. The *type* T of the primitive in I.
2. The *value* W of of the primitive in I.
3. An ordered list of values $\{V_i\}$, one per *parameter* property T_i. Each V_i is the *effective value* of T_i in S_n, where S_n is the subpicture designated by the last name in the path that represents I (Sect. 5.1.3). In other words, each V_i results from propagating the *inherited* value of T_i through the path, from S to primitive.

The precise meaning of "produce" is defined in the context of specific *tracing* methods below.

Rule: An implementation of the Model shall specify the order in which its *parameter types* are processed by the *tracing* methods.

Discussion: Tracing, performing a tree traversal, can be described formally using a *parameter stack*. Let the list $\{V_i\}$ initially contain all *inherited* parameter values in S:

```
Trace(S) {
     Stack {Vi};
     For all parameter properties, taken in the implementation-dependent processing order,
          *Replace the corresponding Vi by its effective value in S;
     For all primitive properties,
          If the corresponding value W in S is not null,
               Produce <T,W,{Vi}>, where T is the primitive type;
     For all child subpictures Z of S,
          Trace(Z);
     Unstack {Vi};
}
```

At the point marked * the algorithm may interrupt tracing S depending on various *effective* parameter values. Examples are if a window has zero extent, or the visibility is off. On interruption, the algorithm skips to the line *Unstack*.

Of particular interest is *tracing* from a *root subpicture*, whose initial $\{V_i\}$ consists of all *default* paramater values. The complete picture, defined by the root, is then traversed (see Section 5.3).

Definition: With reference to the previous definition, ***present*** is a tracing method, where the pair $<W,\{V_i\}>$ produced for each *primitive instance* of *type* T is passed to the *primitive generator* of T (see Discussion).

Discussion:

1. While undefined by the model, a *primitive generator* is meant to be responsible for *presenting* the primitive instance (examples: character generator, circle generator, line generator). It may be implemented, for example, on a graphics card in the hardware. Subject to its capabilities, the generator will achieve this by accessing the parameter values $\{V_i\}$ and the primitive value W according to the definition of the type T of the primitive.

2. An implementation may keep track of the parts (*effective* windows) of a picture that were modified since the latest presentation, and thereby avoid tracing the entire *picture space*.

The *correlate* and *response* methods are described in the next section.

5.2 The Event/Response Formalism

When viewing a picture, the user of an interactive-graphics application is usually confronted with multiple choices on how to proceed. Some of the choices may involve the manipulation of non-graphical devices (e.g. a keyboard), and should therefore, in principle, be of no concern to the Model. On the other hand, one may often handle non-graphic events as special cases of pointings, which may then be integrated into the formalism.

It is intuitively clear that graphic interaction fits excellently with object-oriented principles: the user sends a message to an object (subpicture) by pointing at it. It is therefore natural that the class of *subpictures* provide a (polymorphic) method that has the ability to *respond* to the message. The response may cause an immediate, visual feed-back to the user by updating the *picture space*, or it may perform some non-graphic function. This is all in line with the behaviour of extant windowing systems.

The subpicture object is accordingly vested with both a *syntactic* (visual) and a *semantic* (responsive) role. A programmer, when developing the application with a sufficiently smart editor, could then design the graphics of the subpicture along with its associated response, and perhaps even test them jointly (cf. Section 4, and Soop [1986]).

In a structured picture environment it is useful to say that the user, when pointing at a presented primitive instance, simultaneously points at all subpictures in the corresponding path. In the application, some or all of these subpictures may be associated with responses, which typically need to be executed in path order.

In order to trigger the correct response, the graphics system must then know, for each event, which subpictures the user pointed at (this is the familiar Pick input class in PHIGS). But converting the raw input data from the pointing (usually a device coordinate) into a path is a costly process, demanding in general a complete traversal of the *picture space*, and correlating the pointing with each primitive instance in turn. Therefore many graphics systems leave the operation to the programmer, or at least leave him to decide when to perform it. In my opinion this is not a satisfactory solution. The Model should include a complete *correlate* and *response* mechanism, while providing an escape for those implementations that do not wish to burden the system with the full capability.

5.2.1 Interactive Properties

Definition: After a picture has been presented (on an interactive device), the user may point at it. This action causes an *event* in the graphics system, called ***pointing***.

Rule: A *pointing* is identified by a *coordinate*[8] `K`, expressed in the device system. When the event occurs, the graphics system stores its `K` in an ***event queue***.

Example: A display screen uses pixels in the range [0, 511] in *x* and *y* to address the *canvas*. A pointing in the middle of the canvas is identified by the coordinate `K=`<255,255>.

Discussion: To accommodate workstations with multiple monitors (or windows at the operating-system level) and other interactive devices, the implementation may introduce an extra dimension in `K` that identifies the device. In the above example, the *pointing* might then be identified by the coordinate <0,255,255>, whereas another screen or canvas might yield <1,255,255>, and a ”pointing” at a function keyboard <2,3,3>. A subpicture in the picture space may then be designed to represent a specific device, simply by including a `window` parameter that "clips" away all other devices in the extra dimension.

---

[8]Here, as in the rest of this document, a *coordinate* is defined as an object with *x* and *y* (and possibly additional) components, when Cartesian. Polar and other systems are accommodated in an analogous way.

Definition: The ***access-event*** function is defined as follows:

1. If there is an *event* on the *event queue*, remove it and return it;
2. Otherwise, wait until an *event* arrives, then do (1).

Discussion: The Model is open for events originating from sources other than user pointings, provided they can be described by coordinates in the appropriate range. This mechanism allows timers, sensors, and other sources to emulate user pointings, potentially *animating* the picture.

Definition: By an operation called ***posting*** the application program may put a *pointing* on the *event queue*.

Rule: An implementation of the Model shall provide an API for the *posting* operation. Its argument is a coordinate `K`, expressed in the same range as a *pointing*.

Definition: The *interactive properties* (Section 5.1.2) of a primitive of *type* `T` determine whether or not a given *pointing* ***detects*** a given *instance* of `T`. If it does, the pointing is termed a ***hit***.

Discussion:

1. The precise definition of what is meant by *detecting* is not required by the Model. The design of a particular primitive may implement any scheme that includes hit windows, tolerances, and other criteria. For example, you may provide a circle primitive that can only be detected on the periphery, while another one is detectable in its interior. Note that even if otherwise identical, they are *different* primitives and must be given different *property types* (say, `circle1` and `circle2`) in the implementation of the Model.

2. An implementation may define a *dynamic* parameter (e.g. `detectability`) that disables or enables *hits* in the subpicture.

Definition: With respect to a *hit*, a ***detect element*** is a value that identifies the part of a primitive that was detected. The value may be composite (e.g. an array). The domain and format of *detect elements* is an *interactive property* of the primitive *type*.

Rule: For each *primitive type,* an implementation of the Model shall provide a ***correlation algorithm***, which, given:

1. The *value* `W` of the primitive.
2. A value list $\{V_i\}$ corresponding to all *parameter* properties.
3. A pointing coordinate `K`.

produces either a *detect element* or *`null`* (no-hit case).

Example: An implementation may define a `polyline` primitive with an integer *detect element*, being the index of the line segment that was pointed at. Alternatively, the *detect element* could be an array of indices, to account for the eventuality that the user points at the crossing of several segments.

Definition: A ***hit path*** `D` with respect to a hit with *pointing* `K` is a *path* that represents a primitive instance `I` with respect to a *root subpicture* `R`, detected by `K`, augmented by the *detect element* `d` of the *hit*. In other words, the *hit path* is a list, headed by `R` and ending with `d`:

$$D(R,K) = \langle R, S_1, S_2, \ldots, S_n, d \rangle,\ n \geq 0 \mid Inst(S_n, R)$$

An element (`R`, S_i, or `d`) of a *hit path* is called a ***hit element***.

Example: A pointing may give rise to the *hit path*:

Hit element	Comment
`Pic55`	root subpicture
`Canvas`	drawing window
`Boat`	object being drawn
`5`	polyline segment #5 was pointed at

If two line segments were detected, then the last hit element might be a pair of integers.

Definition: With respect to a *hit*, all subpictures designated by elements in the corresponding *hit path* (including the *detect element* `d`), are said to be *detected* by the pointing.

Definition: A ***hit set*** `{D(R,K)}` is the set of all *hit paths* with respect to a *root* subpicture `R` and a given pointing `K`. A *hit set* may be empty.

Definition: With reference to *tracing* in Section 5.1.5, ***correlate*** is a tracing method, where the pair $\langle W,\{V_i\}\rangle$ produced for each *primitive instance* of *type* `T` is passed, together with a *pointing* `K`, to the *correlation algorithm* of `T`. If the implicit argument is a *root* subpicture `R`, the result is the *hit set* `{D(R,K)}`.

Discussion: The primitive interactive property is a powerful concept that can be used to integrate the various input classes of extant standards into the Model. For example, a Choice device can be described in terms of a *primitive* that yields an integer (e.g. a row index in a menu) as a *detect element*. A Locator device can be represented by a `locator` *primitive* that is invisible on presentation, but covers the local window with "virtual pixels" that can be individually detected by pointing. It then returns the pointing coordinate (pixel address), but transformed to the local system, as a *detect element*.

Furthermore, an implementation of the Model that does not support ”pick input”, may define all primitives, except `locator`, with a zero hit window, hence essentially non-detectable. The application viewport then consists of a single, detectable `locator` primitive. In this way, pointings will yield only coordinate values, and the *correlate* method becomes trivial.

5.2.2 Response Methods

Definition: A subpicture may optionally implement a polymorphic method, ***response***. The default (base-class) method performs nothing.

Rule: The *response* method has one formal argument, which is a *hit element*.

Discussion: A *response* method is normally implemented as a piece of application code, but may also be part of the Operating or Windowing System. It typically uses the *editing* methods to modify the *picture space* between events, but may also perform any other application-related task.

Definition: The ***respond-event*** function invokes all *response* methods associated with the elements of a *hit set* `B`. It may be described formally as follows:

```
respond-event(B) {
    for each hit path D in B,
        for each element E of D in path order,
            if E is the name of a subpicture S {
                let Enext be the next element in D if any, else null;
                invoke the response method of S with Enext as argument;} }
```

Rule: An implementation of the Model shall provide a *respond-event* function and its related API. The argument is a *hit set*.

Discussion: Nothing prevents an *empty* subpicture `S` to implement the *response* method. The method will be invoked if a primitive yields a *detect element* that happens to be identical to the name `S`. For example, a `menu` primitive can be defined to yield the name `Save` when the user points at the choice "Save" in the menu; this would cause the response of a dummy (empty) subpicture `Save` to be invoked, say, to save a file.

5.3 The Dialog

Definition: The ***dialog method*** consists of applying the following three functions in sequence:

		Arguments	Result
1.	*present*	`R`	—
2.	*access-event*	—	`K`
3.	*correlate*	`R,K`	`B`

The method has a *root subpicture* `R` as implicit argument. This object then becomes the implicit argument of *present* and *correlate*. The formal argument of *correlate*, the *pointing* `K`, is the result of *access-event*. Its result, the *hit set* `B`, is the result of the *dialog method*.

Definition: The ***dialog cycle*** is the *dialog method* followed by the *respond-event* function. That is, it consists of the above three functions, followed by:

		Argument	Result
4.	*respond-event*	`B`	—

where the argument is the result of the *dialog method*.

Definition: A ***dialog*** consists of (indefinitely) repeating the *dialog cycle*.

Definition: A ***dialog processor*** is capable of executing the *dialog method* for a given root subpicture `R`. It consists of the following parts, pertaining to a given set of subpicture *property types*:

- A set of *generators* and *correlation algorithms* for its *primitives*.
- A set of *combination algorithms* for its *parameters*.
- A *picture space*, with processors for the *editing* methods.
- Processors to perform the *tracing* methods.
- An *event queue* with a processor for the *access-event* function.

The *picture space* and *event queue* are initially empty.

Definition: The part of the application not in the *picture space* of the *dialog processor* resides in the ***problem processor***. This includes the "traditional" program, in particular, all *response* methods of the application. In addition, the *problem processor* provides the *respond-event* function.

Discussion:

1. The terms "dialog processor" and "problem processor" are used here to denote any mechanism that might be available to separate the functionality discussed. They may be implemented as (physical) micro-processors, or as distinct tasks in a common processor, for example.

2. Several other components can be added to the *dialog processor*, e.g. metafile handlers and facilities to load and save *picture spaces*. Only the parts that handle the basic dialog are listed here.

Definition: An ***edit request*** is produced when the *problem processor* invokes an *editing method* (Section 5.1.5). It consists of either of the following, where `S` denotes a *subpicture*, `T` a *property type,* and `V` a value that may be *`null`*:

Method	Request	Result
query:	the pair `<S,T>`	`V`, such that `<S,T,V>`
set:	the triplet `<S,T,V>`	—

Rule: The *dialog processor* communicates with the *problem processor* through two channels. The traffic on each channel is handled by queues:

Channel 1: A ***response queue*** `Q1`, containing the *hit elements* of a *hit set*. It is directed from the *dialog processor* to the *problem processor*.

Channel 2: Two queues `Q2` and `Q3`, containing *edit requests* and their results, respectively. `Q2`, the ***request queue***, is directed from the *problem processor* to the *dialog processor*, and `Q3`, the ***result queue***, in the opposite sense.

In addition, Channel 1 is used to *post* events to the *dialog processor*.

Rule: While processing the *dialog* (see above), the processors are synchronised by the queues, as follows:

- When needed by the application logic, the *problem processor* (through the *respond-event* function) picks *hit elements* from the latest event off `Q1`. If the queue is empty, the processor waits.
- The *dialog processor* simultaneously senses *hit elements* as they are picked off `Q1`. When the *problem processor* attempts to pick a *hit element* off the empty queue, the *dialog processor* executes the *dialog method*. This puts a new *hit set* on `Q1`.
- The *problem processor*, in the course of processing its *response* methods, queries and updates the *picture space* by putting *edit requests* on `Q2`. For each *query* request, it waits for the result on `Q3`.
- The *dialog processor* services all *edit requests* as they arrive over `Q2`. When the queue is empty, the processor waits.

Discussion:

Initially, all queues are empty, so the dialog processor will be suspended (last point above). Soon *edit requests* defining the initial picture will arrive from the problem processor, presumably as part of application loading. At a certain time the problem processor will want to present its first picture to the user and accept the user's interaction. It will then start trying to pick elements from the *response queue* (second point above), which will trigger the dialog processor to execute its *dialog method*.

After presentation of the initial picture, the dialog processor will most likely hang again, waiting for the user to interact (*access-event* function). The result of the interaction gives rise to new *hit elements* on the *response queue*, which are then used by the problem processor. The dialog will continue in this fashion with two synchronisation points per cycle: one to let the problem processor catch up, one to let the user catch up.

The *dialog processor* may thus be seen as an intermediary, intelligent service component between user and application, providing a dynamic interface between the two.

6. A Possible Hardware Implementation

An implementation of DIPRO can take many forms. It is not difficult to emulate the model by programming (cf. Sect. 3), but there are several aspects of the design that suggest an implementation in special hardware, or at least in special microcode, for selected parts of the system. This special equipment would typically reside in a PC or other intelligent workstation. It would *complement*, rather than replace, conventional processors present in the workstation, and cooperate with these.

The following notes outline some aspects of a hardware DIPRO implementation.

6.1 Processor Layout

Summarising the main principles from the theory (Section 5), we are considering a two-processor split of application execution into a *Dialog Processor*, and a conventional processor. For the latter, performing the purely algorithmic and data management tasks of the application, we have used the term *Problem Processor*.

The communication between the processors takes place over two channels, each transmitting queues (Fig. 1; Section 2.5). Each processor has a memory, but the task of the Dialog Processor being aimed at picture processing, is best served by a specialised storage with its own type of access mechanism, described below. The display functions themselves may be implemented on conventional hardware within the Dialog Processor, such as a graphics processor card, where some tasks, e.g. clipping, hidden-surface cueing, and tree traversal might be assisted by special microcode.

These components of the Dialog Processor are discussed (Fig. 3): the memory containing the *picture space*, its access processor, the main picture-processing unit, and its memory. The graphical devices connected to the workstation and their I/O ports are only mentioned in passing.

6.2 The Picture Space

Following the conceptual design developed in Section 2, the memory of the Dialog Processor stores a set of binary relations, forming the *Picture Space*. In line with Symonds' formalism [1968], the Picture Space `P` is a set of triplets:

```
P: {<R,S,V>}
```

where `R` is a property (relation), `S` a subpicture reference, and `V` the corresponding value[9].

`P` embodies all pictorial information of *one application*, and its contents will in general vary as the application session progresses, causing the picture (or pictures) displayed on the workstation to change. The variation concerns not only the values `V`, but also the set of active subpictures `S`, so the number of triplets in `P` will normally fluctuate substantially during the dialog.

For simplicity we may assume that there is only one graphics device used in the session, and that this device is interactive. Actually, the DIPRO formalism supports multi-device workstations, which may also include passive equipment (printers, plotters, etc., cf. 5.2.1).

As suggested in Section 2.3, one may regard each property `R` as a *binary relation*. Following the relational notation, a given triplet <R,S,V> is present in `P` if and only if the Problem Processor has executed the equivalent of the program statement:

```
R#S = V;
```

where `V` is not *null*, and "#" stands for the access operator (a form of relational *join*). `V` may be a list of values (cf. examples in Section 3.3). On a program reference:

```
R#S
```

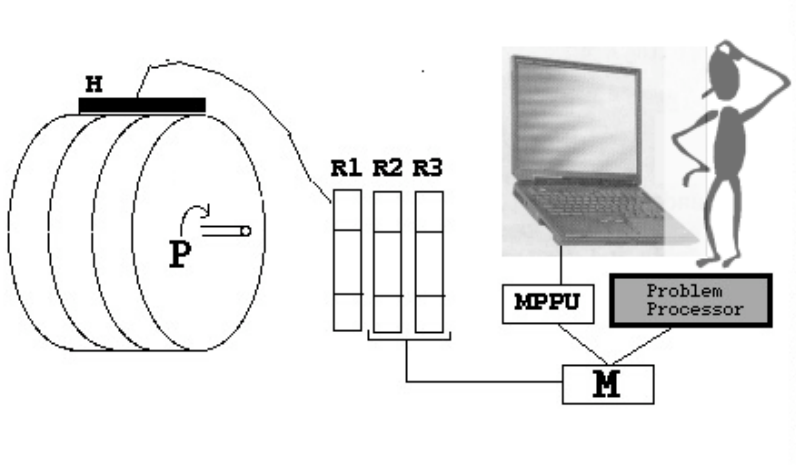
[9] Please note that this section uses a different triplet order from Section 5.

the value (or list of values) V is returned, but *null* will be returned if no matching triplet is present in P.

P is thus addressable only by its contents; being a set it has no internal ordering. Consequently, P is a natural candidate for an *associative memory*.

A convenient model of an associative memory is a cylindrical *drum* [Soop 1980] with one triplet stored along each generatrix (see Fig. 3). A read/write head H, placed along one of the generatrices may then access one triplet at a time. During one rotation of the drum, H will access the entire P; this one-turn operation is defined as the *basic instruction cycle* of the associative machine. Because P has no "first triplet", the cycle may begin anywhere on the drum, and there is no seek delay. Naturally, a physically revolving memory is not suggested (there are several non-mechanical technologies that can be exploited), but the drum analogy is helpful in visualising the basic concepts.

The drum is assumed capable of holding many more triplets than required by the application, and unused triplets are set to a "vacant" value. The width of the drum depends on the representation of the entities R, S, and V. This is explored in the next section, resulting in a typical value of 80 bits for the design example.



P = Picture Space (associative memory)
H = Read/Write Head
R1,R2,R3 = Registers
MPPU = Main Picture-Processing Unit
M = its Memory

Fig.3 Dialog Processor with Associative Memory

In general, the triplets belonging to a particular property R, or to a particular subpicture S, will not be predictably grouped on the drum. After a short period of dialog, they are expected to be scattered around the memory and interleaved with other triplets, including vacant ones, in a pseudo-random way.

6.3 The Memory Words

The triplets are implemented as fixed-length *words* on the drum surface, and each word is divided into at least the three fields corresponding to the entities R, S, and V.

6.3.1 The R Field

As explored in Section 3.3, a general-purpose, medium-function, 2D system will provide fewer than 30 property types. With a suitable encoding, a 5-bit R field might suffice, but as one would like to accommodate a few flags, discussed below, the width may be rounded to 8 bits. This field can simultaneously be regarded as the *op-code* of the machine.

6.3.2 The `S` Field

An application on the level we are aiming at would rarely have more than 200 subpictures at one time, which is probably a high number even for many advanced applications, such as in CAD. By suitable encoding, the `S` field may therefore occupy 8-bit *tokens* that identify the subpictures. The encoding could either take place in the Problem Processor, since the assigned subpicture names are only used at the API level, or it could take place in the Dialog Processor. The latter is preferable, because part of a tokenising mechanism, viz. binary relations, is already present.

We shall therefore assume that the subpicture name is represented by a special property, `name`, whose value is a character string. Encoding then simply means assigning a vacant integer to each new subpicture as it comes along (see Section 6.6.3); this integer is then the required token for the `S` field.

6.3.3 The `V` Field

As might be expected, the `V` field causes more difficulty. Not only are values of varying length, but some of the values are ordered, as shown by earlier examples (e.g. `figure` and `text` in Section 3.3.2). With the drum design and the updating operations described below, it is possible to exploit the inherent order of the words in the storage. Despite the fact that this method compromises the no-order principle of the relational approach, it will be tentatively chosen in this paper. One alternative is to provide an extra sort field per word. Another involves breaking down long primitives into a number of unnamed subpictures, which, however, complicates the updating protocols.

Assuming a single-precision floating-point representation, which is not unreasonable for medium-function graphics, each 2D coordinate would need 64 bits (32 bits *x* and 32 bits *y*). Text primitives may then use the same field to store 8 characters, assuming a standard code (e.g. ASCII) is used. Obviously a "null" code must be reserved for filling purposes.

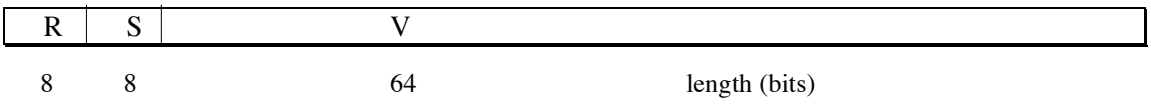


Fig. 4 Word Format

6.3.4 Accommodating the Property Types

The resulting design will have a total drum width of 80 bits (Fig. 4).

With reference to the example in Section 3.3, simple properties, like `pos`, `locate`, and `style`, are accommodated in single triplets, using the `V` field for their numeric quantities. For example, `colour` and `render`, which need to store three or four integers, may split the 64 bits into 16-bit groups to accommodate colour codes and other data.

The `window` parameter carries two coordinates: the upper-right and the lower-left corner. Which triplet is which is signalled by the flag bits in the `R` field. This same technique may then be applied to the more complicated primitives.

The `box` and `circle` primitives are stored in multiple words, whose order is unimportant.

A `figure` primitive may define one or more line sets ("polylines"). Assuming one is always able to store the triplets of the primitive physically in sequence (though perhaps not contiguously), there is only a need to signal the beginning of each new line set (equivalent to the graphic "move" order of some implementations). This is achieved by the same flag as used by `window`.

Text-type primitives (`text, menu`, etc.), are typically implemented 8 characters per word. The first word, carrying the starting coordinate, is identified by the same flag as above.

The state names in `vis` and related parameters could be stored using the same chaining technique as for `text`, but for simplicity (and performance) we assume the names are no longer than 8 characters; or, if they are, they will be truncated. The same applies to subpicture names, stored with the `name` property. In this way, `vis` and `det` are implemented in one or more triplets (the parameters may specify several state names), whose order is unimportant.

`link` has subpicture references as its value part, and significant performance can be gained for this crucial property by referencing through the tokens used by the `S` field. In this way, a `link` triplet may accommodate up to eight links; more are stored in multiple words as required. The order between the links is discussed in Section 5.1.4.

6.4 Relational Access Instructions

Only two basic instructions are defined for accessing the associative memory. They are intended to run on a special micro-processor resident in the main picture-processing unit (MPPU), and are used by various operations described below.

The read-write head `H` of the associative memory is connected to a register `R1`, whose contents are compared for equality with a given word in a register `R2` under a mask stored in a third register `R3` (Fig. 3). The contents of `R2` may also be written (under the mask) to a word on the drum. Registers `R2` and `R3` are connected to the memory `M` of the MPPU.

`get`

> All words that match a given word under the mask are selected from `P` and stored in `M`, starting at a given address. This instruction selects all words `R1` for which `(R1 & R3)=(R2 & R3)`, where "`&`" stands for logical *and*.

`replace`

> Words from `M`, starting at a given address, replace words in `P` under the mask. This instruction consists of two steps. For each word `R1` in `P`:
>
> 1. If `R1` matches `R2` under the mask, it is set to vacant. The selection is the same as for **`get`**.
> 2. If `R1` is vacant (possibly from step 1), `R2` is written, and the next word from `M` is loaded into `R2`. This step is by-passed when no more words remain to be written.

`replace` thus ensures that after an update there will never remain any old words in `P` that match the selection.

These instructions nominally take one machine cycle. If, for timing considerations, it is not possible to read and write the same word as it is accessed, the design may have to incorporate separate read and write heads. Also, it may be necessary to perform the two steps of **`replace`** in separate cycles.

The only logical error that can occur is that the drum gets full during **`replace`**. Full reset of `P` is achieved by **`replace`** with a zero `R2` and `R3`.

6.5 Tracing Operations

Tracing is the fundamental operation of the Dialog Processor. The objective is either picture presentation or correlation, i.e. phases 1 and 3 of the Dialog Cycle (Sect. 5.3). It is a well-known task in structured graphics (see e.g. Foley & Van Dam [1982]). Only a brief description is given here, adapted to DIPRO.

Tracing involves traversing the tree of the picture space, starting at a given root and visiting all descendant subpictures in hierarchical order. It is a recursive operation, whose state is maintained by a special set of registers, here jointly called the *Graphic Status Word* (GSW), in analogy with the PSW of many

conventional processors. The GSW contains the current effective value of *all parameters* (Sect. 5.1.3), in addition to the local subpicture token. It is supported by a stack in the memory `M` (cf. Sect. 5.1.5) capable of holding as many GSWs as there are levels in the structure (if a limit is set by technology, it should be at least 20). All coordinates in the GSW are expressed in the *device* system (i.e. the system used by the primitive generators of the graphic device); they may therefore be integers, typically representing a pixel index. Before a tracing operation starts, the GSW contains the parameter defaults (Section 5.1.3).

During tracing, the Dialog Processor calls on several important functions, some of which may involve hardware or firmware. For example, when a primitive is encountered during Presentation, it must be sent to a *generator*, which is often a hardware unit producing the only tangible result of tracing, viz. display. However, if encountered during Correlation, the primitive is compared with the user's pointing, a function that may be implemented in software. The tracing operations also use a *clipper* for windows, and a *transformer* for coordinates, which may be implemented in microcode or in software. These units all need access to parameters in the GSW and are included in the MPPU, where tracing takes place.

The relational triplets of a given subpicture must be traced in a certain order. First, it is necessary that all parameters be traced and dealt with before any links or primitives, since the latter should be presented *under the effective* position, colour, etc. Secondly, depending on the geometric transformations supported by the implementation, it may be necessary to perform, e.g. positioning before scaling, and window clipping after all other geometric parameters. This order is preserved by implementations of the Model (Section 5.1.5).

During tracing it may happen that the current subpicture is abandoned because of certain parameter combinations. Typically this occurs on a non-matching visibility parameter, but it may also be caused by the effective window disappearing, or for other reasons. To speed up the operation, it is therefore desirable to put `vis` and allies among the first in execution order (i.e. assign a low op-code). In the following, some of the subpicture properties from Section 3.3.3f will be used as examples.

6.5.1 Presentation Trace

Tracing a subpicture, identified by token `S`, for Presentation is achieved by the following algorithm:

1. **`get`** all words `W` of `S`, except `link` and "long" primitives like `figure` and `text`. The instruction involves a mask on the `S` field and part of the `R` field.

2. Sort `W` on the `R` field. Practically, this means that a sort index is generated on the addresses in `M` where `W` is stored.

3. Process `W` in the sorted order. In particular:

   - On `vis`, if none of the states matches the GSW state, abandon the tracing of `S` altogether.

   - On `name`, stack the GSW, and place the `S` token in the GSW.

   - On `pos`, `orient` and `scale`, combine the `V` field with the corresponding register in the GSW and replace the result in the GSW. For `pos` this means passing it through the transformer.

   - On `window`, transform the coordinates in the `V` fields, then clip them and replace the result in the GSW. Then, if the window is of zero extent, abandon the tracing of `S` by branching to step 6.

   - On `colour`, `style`, `font`, `render`, and `state`, replace the parameter in the GSW by the `V` field.

   - A "short" primitive (e.g. `circle` or `box`) is passed through the transformer and the clipper, then sent to the hardware generator. The generator also needs effective parameters from the GSW, such as line style, colour, and rendering.

   - `locate`, by definition, uses the effective window from the GSW and needs no further transformation or clipping. Subject to its value `V`, compute and send coordinates representing the window and grid to the line generator.

- `key` and `poster` are ignored altogether.

4. **`get`** each "long" primitive of `S` in turn from `P`, overwriting the earlier `W` in `M`. The instruction involves a mask on the `S` and `R` fields.

   - The coordinate of a `figure` word is transformed, then clipped before being sent to the line generator. In addition, contingent upon the buffering capability and intelligence of the latter, the algorithm may have to maintain a current position which is updated by each word in a line set.

   - `text` (and equivalent primitives) also makes use of a current position. The text generator uses the effective `font` and `colour` parameters from the GSW.

5. **`get`** all `link` words of `S` from `P`, overwriting the earlier `W` in `M`. The instruction involves a mask on the `S` and `R` fields. Trace all subpictures referenced in the `V` field. During this time, the current `W` is preserved.

6. Unstack the GSW, i.e. replace it from the top of the stack and remove the top.

Some subpictures may be very large due to long line sets or text strings. In an extreme case, the picture consists of one very large line drawing. To avoid a correspondingly large buffer in `M`, the latter may be defined as a wrap-around area, so that, racing conditions permitting, the "long" primitive is processed in parallel with the **`get`**.

This algorithm is executed with the token of the root subpicture, representing the total presentation area of the device, as the initial `S`. As the window of the root is processed, the clipping algorithm will sense the infinite default window in the GSW and perform the mapping onto the physical system of the device or *canvas*, by updating the `pos` and `scale` parameters in the GSW (Section 3.2.3). If required by the hardware, the previous picture is erased and a new picture is initiated at this time, e.g. on a printer the next page is loaded.

In principle, the order in which primitives and links are processed is irrelevant for presentation. Due to device technology, however, the visual effect may depend on that order, especially if rendering is used (cf. Section 5.1.4).

The stacking mechanism guarantees that the final contents of the GSW are identical to its initial contents, so there is no need to refresh it before the next tracing operation begins.

6.5.2 Correlation Trace

Tracing for Correlation follows a similar algorithm. The main difference is that primitives are not sent to generators, but are compared with the pointing of the current event. This event was accessed in the intervening Access phase (Section 5.2.1). Each primitive type has in principle its own *correlation algorithm.*

If the algorithm finds that the pointing (expressed in the device coordinate system) lies on, or in some cases in, the primitive (within a certain tolerance), it generates a *hit*. On a hit, the path consisting of all `name` tokens in the GSW stack (including the current GSW itself) is copied over to a free area in the memory `M`. In addition, the primitive itself (through its algorithm) generates an ultimate *detect element* for the path; this may be a number, a coordinate, or a character string (cf. Sect. 3.3). At the end of tracing, all tokens in the path are converted to names, and the resulting Response Queue is made available to the Problem Processor. Because the user can point simultaneously at several primitive instances, the operation cannot just stop on the first hit. Each hit generates its own path, which contributes to the Response Queue.

Other differences occur in the individual treatment of some parameters and primitives. Detectability, being ignored during Presentation, is here treated as `vis`. Most cosmetic parameters may in principle be ignored altogether. `window` terminates tracing of the current subpicture immediately if the pointing lies outside the transformed window. This, in turn, allows `locate` to always generate a hit without further checking. `key` is analogous, generating a hit if the pointing indicates that one of the specified keys was pressed.

Also here the sequence in which links are processed may be important for the result. Though undesirable in a well-structured application, it cannot sometimes be helped that the response program depends on the order of Response Queue elements. Therefore the tracing algorithm (step 5), should process the links in the order they have been specified to the Dialog Processor.

6.6 Update and Reference Requests

The theory of DIPRO makes the *subpicture* the basic, in fact the only, unit of pictorial information that can be addressed by name. The application designer is encouraged to create new subpictures, rather than trying to cram structure into sets of primitives, and typical applications are expected to flood the Picture Space with subpictures, some of which may even be generated automatically by the API.

Creation and destruction of subpictures are required to be not only easy and comfortable at the API level, but fast to execute. In fact, there should be next to no extra overhead to pay, if a subpicture to be "updated" does not yet exist in the Picture Space.

In this vein, the following operations, utilising the *Request Queue*, are defined (cf. Section 2.7):

6.6.1 Subpicture Update

The property name is encoded for the `R` field by a simple table look-up function. If it is invalid, an error is returned on the request channel. The subpicture name of the request is tokenised, and if the name is not present in `P`, a `name` word is added (see below).

Then the update is made through the **`replace`** instruction with the mask on the `S` and `R` fields. Note that *all* triplets associated with the `S` and `R` must be replaced; it is not possible, for instance, to update only one character in a `text` primitive, or to add only one coordinate to a line set. Since the operation takes one machine cycle anyway, this is immaterial.

If no values are provided (the value part is *`null`* in the Request Queue), the operation sets all matching triplets to vacant.

6.6.2 Subpicture Reference

The property name is converted as above. The subpicture name is tokenised, and if the name is not present in `P`, *`null`* is returned on the request channel. Else the reference is made through the **`get`** instruction, and the result (which may be *`null`*) is returned.

6.6.3 Tokenisation and Verbalisation

To tokenise a subpicture name, **`get`** the `name` word in `P` that specifies the requested name with a mask on the `S` and `V` fields. If the name is found, return the token in the corresponding `S` field. Otherwise, if an updating operation was the caller, use **`replace`** to add a new `name` word with an unused token, and return this token.

The opposite operation, verbalisation, consists of **`get`**ting the `name` word with the given `S`.

6.6.4 Other Operations

Although not discussed in the earlier sections, there must be ways to manipulate complete subpictures through the API. The basic operations are fetch, delete, and replace a subpicture. All are achieved by several **`get`** and **`replace`**, with a mask on the `S` field. The subpicture name is tokenised as above, and the traffic uses the request channel.

A *post* request, consisting of the coordinate of a simulated pointing, may arrive from the Problem Processor. This event is enqueued (see below), and used in lieu of a user-generated event in animated presentations.

The Dialog Processor may be *booted* from the Problem Processor through a special request. The entire `P` is then initiated to vacant words (through **`replace`** with a zero mask), the GSW is set to default parameter

values (the `name` token in the GSW is set to zero), and all pending events are cleared. After this the Dialog Cycle (see the following section) is started.

6.7 Dialog Cycle Execution

For completeness we recapitulate the way the Dialog Cycle is executed according to the Model developed in the earlier chapters.

With reference to Sections 2.4 and 5.5, the main process running in the Dialog Processor executes the first three phases of the Dialog Cycle. This process services three queues: the *Response* and *Request Queues*, communicating with the Problem Processor, and an *Event Queue*.

Each user interaction as well as any other event, such as sensor input, time-out, or a post, is enqueued on the Event Queue as a coordinate, expressed in the device system.

The interplay between the three queues serves to synchronise the Dialog and Problem Processors. The Dialog Processor performs the following algorithm, where we retain the original numbering of the phases (Section 2.4), but begin with the Response phase. This phase was earlier described in terms of the Problem Processor, but we are now concerned with the corresponding task of the Dialog Processor:

4. **Response**

   Pick requests off the Request Queue in FIFO order and service them, while monitoring the state of the Response Queue. During this time new requests may arrive from the Problem Processor; these are dequeued and serviced. At the same time, the Problem Processor will pick elements from the Response Queue as part of its normal application processing. Each such pick is sensed by the Dialog Processor. Now, when the Response Queue is empty, as soon as the Problem Processor attempts to pick the next (non-existing) hit element, stop servicing the requests, and execute the next phase.

1. **Presentation**

   Run the Presentation Trace. The user will now view the complete updated picture.

2. **Access**

   If the Event Queue is empty, wait for an event to arrive. Dequeue the first event, and mark it as the current event.

3. **Correlation**

   Run the Correlation Trace using the current event. The resulting *hit set* is transmitted to the Problem Processor via the Response Queue.

Then repeat from Phase 4.

As mentioned in Section 5.3, the Response Queue is initially empty and ignored by the Problem Processor, so the Dialog Processor will be suspended in phase 4. During this time, update requests defining the initial picture will arrive from the Problem Processor, presumably as part of application loading. At a certain time the Problem Processor will want to present its first picture to the user and accept the user's interaction. It will then start trying to pick elements from the Response Queue, which will trigger the Dialog Processor to go on with the other phases.

After presentation, the Dialog Processor will most likely hang again in phase 2, waiting for the user to interact. The dialog will continue in this fashion with two synchronisation points per cycle: one to let the Problem Processor catch up, one to let the user catch up.

The Dialog Processor may thus be seen as an intermediary, intelligent service component between user and application, providing a dynamic interface between the two.

7. Conclusion

The model outlined in this paper has been implemented by the author and extensively used, resulting in a dramatic boost in expressive power and productivity for application design. This comes in part from the extreme simplicity and economy of concepts — the implemented API has fewer than 30 entry points (to be compared with over 500 for PHIGS, for example). A second important reason is the object-oriented base for the Model, leading to a formalised, yet natural coupling between picture and program.

References

Foley J.D. & A. Van Dam [1982]: Fundamentals of Interactive Computer Graphics — Addison-Wesley.

ISO [1990]: The Computer Graphics Reference Model — ISO/IEC JTC1/SC24/N512.

Minsky M. [1975]: A Framework for Representing Knowledge, The Psychology of Computer Vision (P. Winston, Ed.) — McGraw-Hill Book Co., Inc., New York: 211.

Palermo F. [1979]: Some Database Requirements for Pictorial Applications (Lecture Notes in Computer Science 81) — Springer: 555.

PHIGS [1985]: Programmer's Hierarchical Interactive Graphics Standard — ANSC X3H3/84-44.

Schauer U. [1983]: The Integrated Data Analysis and Management System (Lecture Notes in Computer Science 150) — Springer: 30.

Sharman G. [1979]: A Picture Drawing System using a Binary Relational Database — Data Base Techniques for Pictorial Applications, Springer: 425.

Soop K. [1980]: A Data Support System — IBM Nordic Lab, TR 18.228.

Soop K. [1982]: A Graphics Dialog Model — IBM Nordic Lab, TR 18.229.

Soop K. [1986]: Bringing Graphic Dialogs to APL — Proceedings of the APL86 Conference, Manchester: 96-102.

Soop K. [1988a]: APL Graphics and the Associative Machine — Proceedings of the APL88 Conference, Sydney: 306-313.

Soop K. [1988b]: Picture Structuring in Man-Machine Dialogs — Proceedings of the SEAS AM88 Conference, Aalborg.

Soop K. [1989]: An APL Model of a Graphics Dialog Processor — IBM Nordic Lab, Sweden.

Soop K. [1992]: APL and the Graphics Programming Interface — APLCAM.

Symonds A. J. [1968]: Auxiliary-Storage Associative Data Structure for PL/I — IBM Systems Journal 7,3: 229.