Graphics in the Views system

L.G. Barfield

Computer Science/Department of Algorithmics and Architecture

CS-R9260 1992

Graphics in the Views System

Lon Barfield

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Lon.Barfield@cwi.nl

Abstract

The Views system is a new computing environment currently under development. One of its many facets is a graphics facility enabling structured picture objects to be easily entered and manipulated. Such pictures can be included as illustrations in text, as icons in the user interface and ultimately as the specification of all the visible aspects of the system; menus, scroll-bars, windows and so on. Views graphics objects can also be linked to other Views objects with invariants, resulting in pictures that automatically react to changes within other parts of the system. This document outlines the Views system and describes some of the design and implementation issues of the graphics facilities.

The
VIEWS
System

1991 Mathematics Subject Classification: 68U05, 68N99, 68P05.
1991 CR Categories: I.3.0, I.3.4, I.3.6, H.5.2, H.1.2.
Keywords and Phrases: computer graphics, graphics utilities, graphics methodology, ergonomics of computer software, user interfaces, data presentation.

1 The Views System

Views is the name of a new computing environment being developed at the CWI in Amsterdam. It follows on from the ABC project [1], which was concerned with the design of a new programming language and dedicated editor. The main design goals of ABC were oriented towards the end user, the language was to be:

- Simple.
- Obvious.
- Interactive.
- Well unified.

With the Views project these same goals were applied to the design of a more general computing environment. An environment which was simple enough to give users without a computing background access to powerful computer systems, an environment which was highly interactive and predictable in its behaviour and where the concepts and models learned in one context could be easily and safely applied to tasks in other contexts. In short, an environment to promote the usability of all aspects of the computer system and to guarantee the usability of extensions to the system [2].

1.1 An example

The best way to explain the Views system is through a demonstration. It is only then that an audience really appreciates what the system has to offer. As we are restricted here to text and diagrams, we shall ‘talk through’ a simple session with Views. Throughout this example the explanation will be kept to a minimum, the detail will be covered in the following sections of the paper. We will assume that the user wants to edit some scripts (Views ‘programs’) and demonstrate some simple graphics.

Upon starting Views the user is faced with something similar to Figure 1. It is a view of an object called ‘Home’ and it contains several other objects; ‘Pictures’, ‘Scripts’ and so on. The user wants to create a similar view of the contents of the object called ‘Scripts’ which contains scripts describing Views objects and invariants. At the moment the focus (containing those objects that will react to the users commands) is on all of the objects in ‘Home’. This is shown by them all being underlined. The user is able to issue commands that alter the scope of the focus or affect the focus contents. These commands can be issued through the use of menus or key strokes. In our example the user is able to alter the focus using the command ‘first’, this narrows the focus down on to the first of the

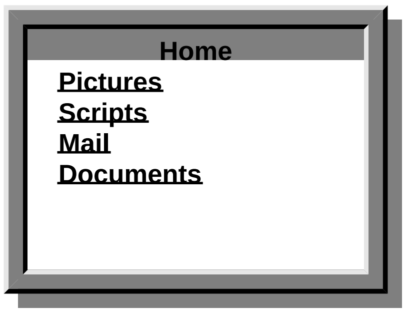

Figure 1 The start of a Views session.

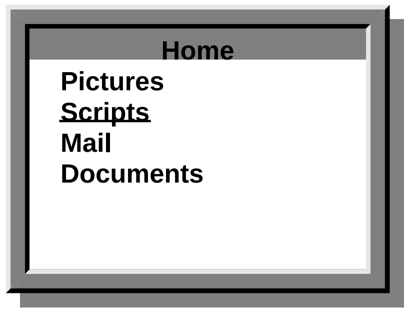

Figure 2 Moving the focus.

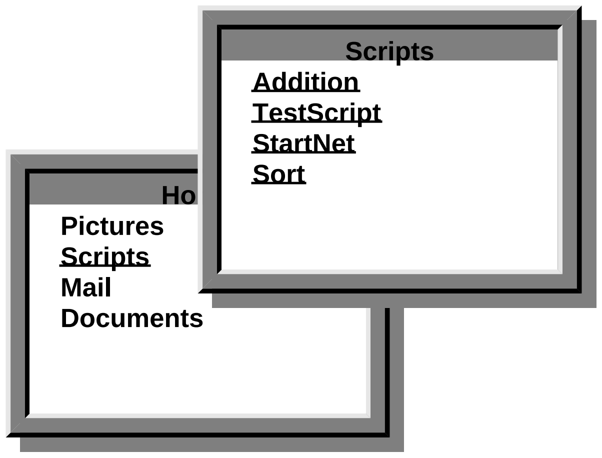

Figure 3 Visiting the Scripts object.

objects currently in the focus (in this case onto ‘Pictures’), they may then use the command ‘next’, moving the focus onto the following object (‘Scripts’). The situation now is shown in Figure 2.

To create the windowed view of the ‘Scripts’ object the user may now use the ‘visit’ command. This acts on the object in the focus displaying its name and contents in a separate window, (see Figure 3). In appearance and behaviour it is the same as the ‘Home’ object, the only difference is the contents.

In a similar manner the user can now narrow the focus to the ‘Addition’ object and visits that. They may also close or iconise the ‘Home’ and ‘Scripts’ views, resulting in Figure 4. Once again it is a case of: same appearance, same behaviour, different objects. In this case the contents are numeric values (2, 3) numeric values shown as textual labels (a, b, c) and invariants between them (+). This script is not just composed of ‘dead text’, it is a description of system objects, and invariants connecting those objects. The letter ‘a’ here is more than a character, it is a textual label for a numeric object within the system that has a numeric value. Using the operators described above, the user now visits ‘b’ and then ‘a’ resulting in Figure 5.

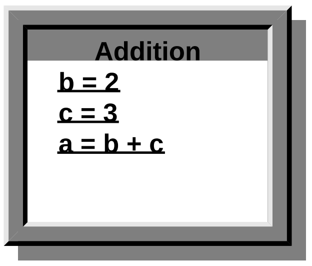

Figure 4 Visiting the Addition object.

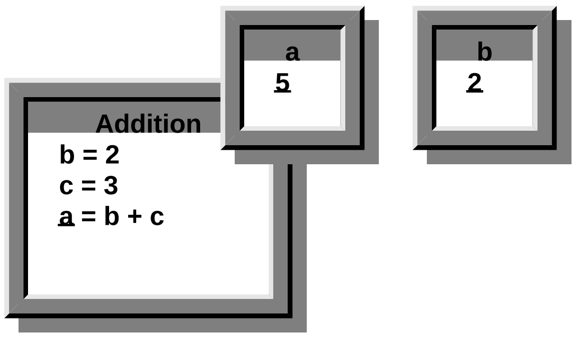

Figure 5 Visiting objects ‘a’ and ‘b’.

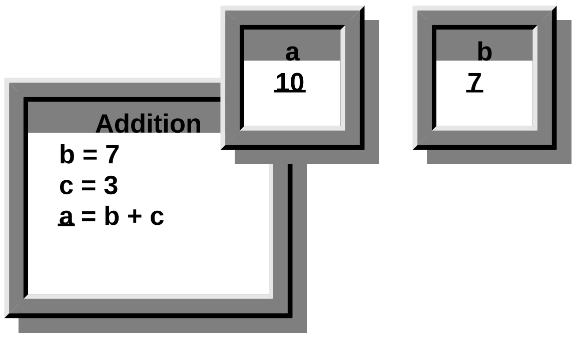

Figure 6 The results of editing an integer value.

Turning to the contents of the focus in the ‘b’ window, (the numeric value 2), the user edits this by deleting the 2 and typing in a 7. The system reacts in two ways to this change: firstly the 2 in the script also changes to a 7, since it is the same object. Also the addition invariant (+) between a, b and c causes the value of ‘a’ to change to 10 (a=b+c i.e. 3+7). See Figure 6.

The user now edits the script to include an example of a graphic object; the function ‘Rectangle width height’, which returns a rectangle with the given dimensions, in this case a and b. They then visit this new rectangle and it is displayed graphically in a separate

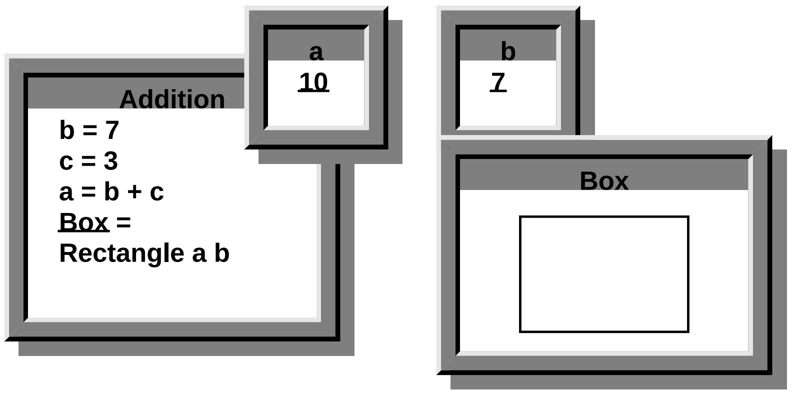

Figure 7 An example of a graphic object.

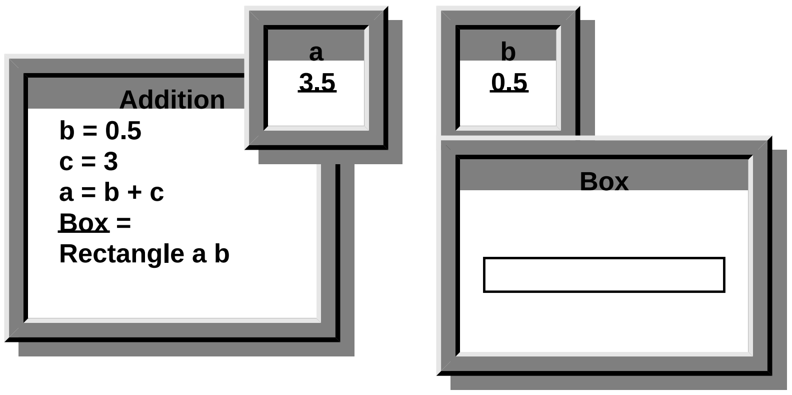

Figure 8 Editing the parameters of a graphic object.

window; see Figure 7. Now if the user edits the value of ‘b’, changes occur in the script, in the value of ‘a’ and in the form of the rectangle, all of which are dependent upon the value of ‘b’. See figure Figure 8.

1.2 Views objects

From the above example one can see that there are three basic components in the Views system: the underlying objects that make up the system, the invariants between these objects and the editor that enables the user to manipulate the objects. We shall look at these in a bit more depth.

Everything within Views is an object of some type. Everything from the smallest number to the most complex structure. These objects can be of an atomic type (numeric values, characters), or of more complex structured types (scripts, names, coordinates, words) involving sub-objects of other types.

Complex objects are tree structured with each object having the potential to have any number of child objects as sub-objects. Thus a two-dimensional coordinate could be modelled as a complex object with two children (representing the X and Y values) each of which is a numeric object. Ordinary text can be built up from a hierarchy of objects; a text document is made up of sections, a section of paragraphs, a paragraph of sentences and so on. Within the Views object model it is possible to have an object occurring within more

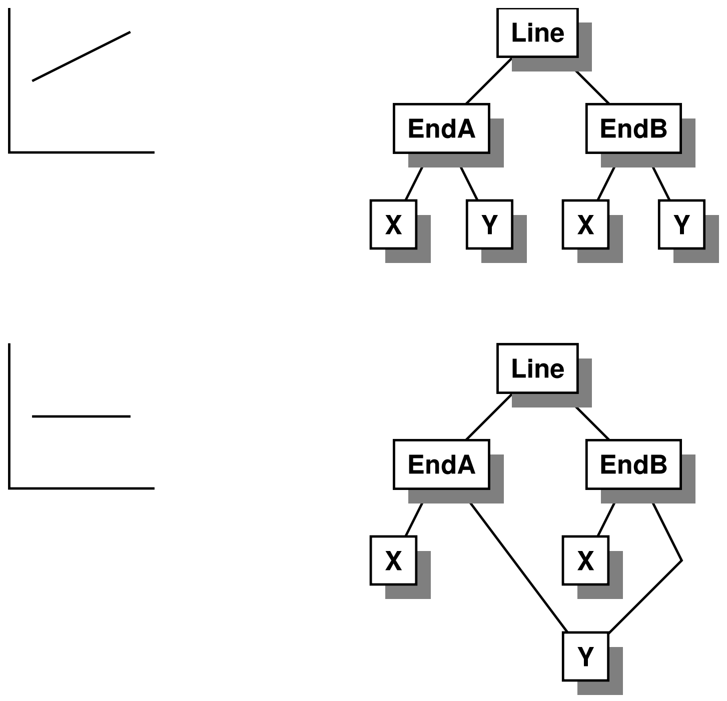

Figure 9 A generic line and a constrained line.

than one complex object. Thus a line could be constructed with each of its two end points having the same numeric object present as the Y coordinate; one object occurring in two places. This would constrain it to always be parallel to the X axis, (see Figure 9).

1.3 The general editor

The general editor is a syntax directed editor which can be used to edit *any* object within the system. As was shown in the opening section whenever you ‘visit’ an object it pops up on the screen in a window where it can be edited using the suite of editing commands.

All user actions in the system are carried out by editing objects through this editor regardless of what the object is. Thus, this editor is the user-interface to the whole system, the same delete command is used whenever you want to delete something regardless of what it is that is being deleted. It could be a mail message, the name of a news group you want un-subscribed, a process to be killed, a sentence in a document, a circle in a diagram, anything.

An essential part of the editor is the focus. This is the ‘area of interest’ or ‘area of operation’ that encompasses some part of the object. The user has two groups of commands; those that alter the scope, or extent of the focus, and those that perform some operation upon the objects that are currently in the focus. Both featured briefly in the opening sections, a detailed description of the focus move commands is not necessary, it will suffice to say that using them it is possible to bring any combination of the sub-objects of an object into the focus. The commands that operate on the focus contents are described below.

- **Delete:**
  This deletes the objects in the focus. Since the contents of the focus ‘disappear’, the focus is left with nothing in it and is effectively invisible to the user. Thus it is repositioned on an adjacent object to the one that was deleted. For example deleting a word of text would result in the focus being repositioned onto the previous word.

- **New:**
  New creates an object. The type and position in the structure of the new object are decided upon by the system according to the current position and contents of the focus. For example if the focus were on an object that was part of a sequence then ‘new’ would create a new object within the sequence located after the object that was in the focus.

  As well as the ‘new’ command the user is able to create some types of objects by typing text directly into the system. The text is interpreted by the system according to its syntax and context and new objects and invariants are built up accordingly. Thus when typing a text document in, the user does not need to be concerned with the construction of the underlying structure.
- **Copy:**
  This copies objects into a copy buffer. Its behaviour is a little more complex than at first appears. This is because ‘copy’ can mean three different things in the context of the Views system; copy instance, copy object or copy both the object and the invariants. Copy *instance* means that another instance of the same object is placed in the copy buffer. If the original object is edited this second instance also changes (since it is in fact a reference to the same object). Copy *object* results in another object with the same structure as the first but not having any of the invariant links to other objects that the first had. Copy *object and invariants* results in a structurally similar object (as above) but this time it is ‘live’ in that it includes the invariant links to other objects that the first had.
- **Paste:**
  Paste moves the objects from the copy buffer into the object being edited. As with ‘new’, the position of the new objects are decided upon according to the current position and contents of the focus.
- **Visit:**
  This command pops up a new window showing the contents of the object in the focus.

The suite of focus scope commands and focus contents commands is available to the user through pull down menus or associated key bindings. It should be mentioned that, in the spirit of Views, the on screen menus themselves are just one view of an internal Views object. This object can be visited and edited to produce immediate changes in the appearance and behaviour of the menus. The text appearing in the menus can be changed, the menus can be reorganised, key bindings edited and newly written functions can even be pasted directly into them.

As well as controlling the scope of the focus by issuing commands, the user may also use the mouse to manipulate it; clicking on an object in a view to position the focus on it, or fixing one end of the focus and dragging the other to include several objects.

1.4 Invariants between objects

The dynamic and reactive nature of Views is achieved by the specification of invariants between objects. If one object changes it produces changes in other objects that are linked to it by invariants and so on, producing effects that ripple through the network of objects that make up the system.

A simple example of an invariant is the plus operator in the preceding ‘talk through’. The contents of object ‘a’ are constrained by the invariant to be equal to the sum of ‘b’ and ‘c’. If one or both of them should change then ‘a’ will be changed by the system in order to maintain the invariant.

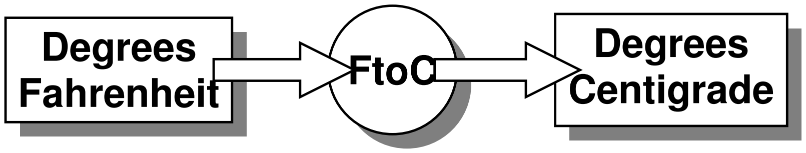

Figure 10 The Fahrenheit to Celsius invariant.

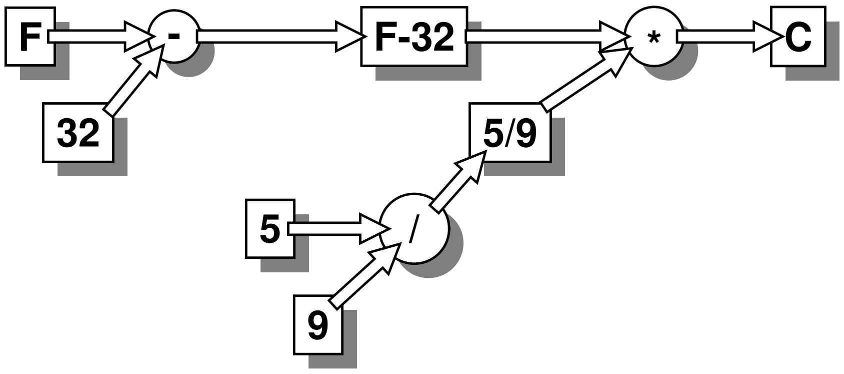

Figure 11 The network of basic invariants.

The Views system includes a suite of basic invariants that can be used as 'building blocks' to assemble more complex invariants. These invariant links are specified by the use of a functional language. Thus if someone were constructing a degrees Fahrenheit to degrees Celsius invariant to connect two objects, one representing Fahrenheit and the other the same temperature but expressed in degrees Celsius. They would build it out of a number of simple mathematical invariants and numeric objects of constant value. The script to construct the invariants would look something like this:

FtoC(F) =(F - 32) * 5 / 9

and be used thus:

A = FtoC(B)

The simple conceptual view of this invariant would be that shown in Figure 10, but the actual network of invariants from which it is made is shown in Figure 11. This construction of invariants from small basic invariants makes the Views system more efficient in the following respects:

- **Efficient multi-tasking:**
  When an object gets changed, all the invariants that need to be processed are queued and treated sequentially. If the invariants were monolithic and large then processing one would clog up the system for a time and prevent anything else from happening. For example if two monolithic invariants A and B were in the queue, first A would be processed and after that B. Having many basic invariants makes the system behave in a multi-tasking manner since the sub-invariants become interleaved on the queue as the processing of A and B takes place. Thus, if A were broken down into the sub-invariants A1, A2, A3... and likewise for B, the queue could look something like this; A1, A2, B1, A3, B2, B3, A4, B4, B5, B6, A5...

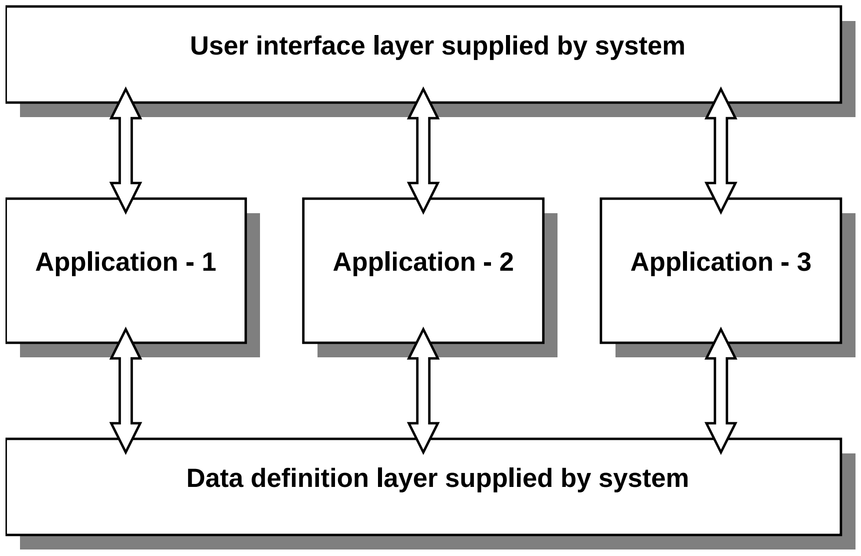
User interface layer supplied by system
Application - 1
Application - 2
Application - 3
Data definition layer supplied by system

Figure 12 The user-interface and data definition layers in Views.

- **Optimal processing:**
  With a large and complex invariant between several objects a change in one of them will result in the processing of a large and complex function to update the other objects. When such an invariant is broken down a similar change may instead cause just a few of the sub-invariants to be processed.

  A good example is a matrix multiplication invariant multiplying two matrices (A and B) to update a third (C). Each matrix is a complex object containing a number of numeric sub-objects. If it is implemented as one large invariant a change in one of the numeric sub-objects of A say, causes an entire matrix multiplication operation to be carried out. If it is broken down into a network of simple addition and multiplication invariants then the same change will cause just a few of the invariants to be processed and only the sub-objects in C directly affected will be updated.

1.5 Views applications

The resulting system is one that provides both a data definition layer underlying all applications, and a general user-interface layer above all applications. This approach guarantees a consistency of data organisation and interchange throughout all services and applications, and also a consistency in user-interface behaviour across the entire system. Figure 12 shows the organisation of the layers in Views, contrast this with the more conventional philosophy where each application must take care of its own data definition and user interface, see Figure 13.

Writing a Views application is thus a relatively simple task. The author does not need to be concerned with the detailed design, behaviour and programming of the user interface. Similarly they do not need to bother with any of the issues of the data structures; designing them, storing them, writing them to files, ensuring compatibility with other applications. All of this is already there. All the author has to do is to write the bit in the middle; the *real* application program, and it will automatically be guaranteed to conform to the systems data handling and user-interface behaviour. Not because the application writers must follow a list of guidelines and not because they must make inclusions from a ‘tool-box’, but because both the data handling and user-interface behaviour are *integral*

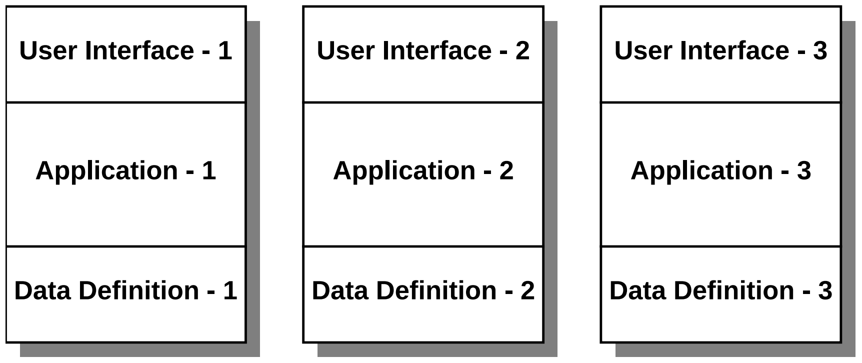

Figure 13 Separate user-interface and data definition layers.

parts of the Views system. Furthermore, it should be remembered that a Views ‘application program’ is not just a list of instructions to be obeyed one after the other in the conventional sense of programming, instead it is a specification of a network of objects and invariants through which changes can propagate in response to the user editing certain objects.

The language used for writing the scripts will be Squiggol; an advanced functional language that enables the functionality to be expressed accurately and concisely, and assists with the breaking down of the invariants into basic invariants [3].

1.6 Presentation of objects

Another important facet of the system is the mechanism for the presentation of objects on the screen when they are being visited. Each object of a particular type contains only the abstract structural information and the values for that object. There is no indication of how it should be displayed. That information is contained within a separate object; the associated display grammar object. For example a two-dimensional coordinate object could be made up of two numeric objects (NUM1 and NUM2). The object itself contains only these two numeric objects. Its associated display grammar describes how to display these numbers, for example:

"(" NUM1 "," NUM2 ")"

The items enclosed by quotes are literals and are displayed as they stand, while the two identifiers are displayed according to their own display grammars. As they are both numeric objects the display is simply a textual representation of their value. Some objects may have several display grammars from which the user can select the most appropriate for the situation. It is thus possible to visit an object in several different ways at the same time, and any changes in the underlying object will be seen automatically in each of the different views. A sequence of numbers, for example, could be viewed as a normal line of numerals, a textual presentation (‘two’ in place of ‘2’) or as a barchart. Editing the underlying sequence, which can be carried out through either of the views, results in all three views being updated automatically.

The display of objects on the screen is achieved, like everything else, by the specification of invariants between objects. In this case between the object being displayed and the view of it on the screen. Any change in the object is immediately reflected by a corresponding change to its view. This display invariant is also connected to the display grammar of the displayed object. The situation is shown in Figure 14. Since the display grammar is a Views object it too can be edited by the user resulting in the layout of the view changing.

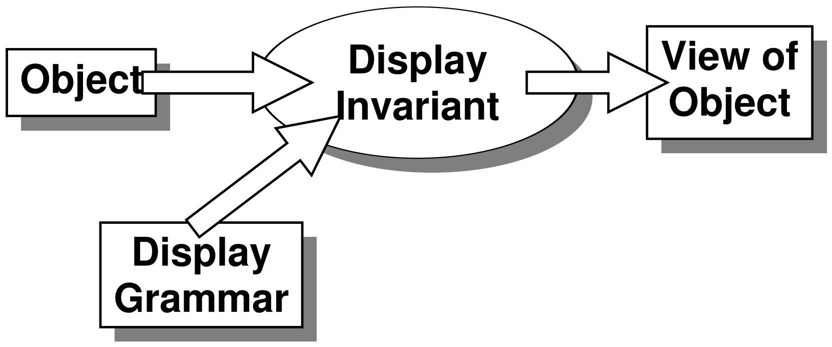

Figure 14 The display invariant.

1.7 Incremental presentation

The Fahrenheit to Celsius invariant described in Section 1.4 is an invariant between two objects that is in fact made up of many small invariants. Contrast again Figure 10 and Figure 11. In a similar way the display invariant described above can also be made up of many small invariants, each linking a sub-object to the patch of the screen that it is displayed in.

Due to this, changes in the object being viewed do not need to cause a complete redraw of the view, but only a redraw of those areas directly affected by the change. Thus deleting a character in a textual object may only cause the redisplay of a section of one line. Inserting characters could cause the line to ‘spill over’ onto the line below thus causing an update which, although it may effect several lines, is still partial.

2 Graphics in the Views model

Graphics is an integral part of the Views model. Using the functional specification language it will be as easy to specify a pictorial view of an object as it is to specify a textual view. This will be of obvious benefit to such things as illustrations within text and visualisation of data. Furthermore the ability to incorporate dynamically changing objects within a picture object results in ‘animation for free’ within the Views system. Ultimately the distinction between text and graphics (words and pictures), breaks down and text can be modelled as a part of the graphics. This leads to another important advantage in that the entire on-screen appearance of Views; text, windows, fonts, scroll-bars, menus and so on can be described within the system as Views picture objects.

2.1 Displaying picture objects

The display of picture objects follows the display of other objects within Views. They can be displayed using a choice of display grammars, either as a textual representation of the structure or as a two-dimensional picture, see Figure 15. In the latter case the display invariant involves an extra object; a description of the rectangular area within the window that the picture is to be placed in. This allows it to be scaled and positioned correctly.

If the picture is the only thing being displayed in a window then this rectangular area is formed by the boundaries of the window. Should the picture be included in another object, a text document say, then the rectangular area will be some portion of the window depending upon which part of the document is being viewed. As with any other Views object, changes to the structure are reflected immediately in the view of the picture. Such changes can be a result of direct editing of the picture object by the user or of invariant links to other objects in the system that may be changing.

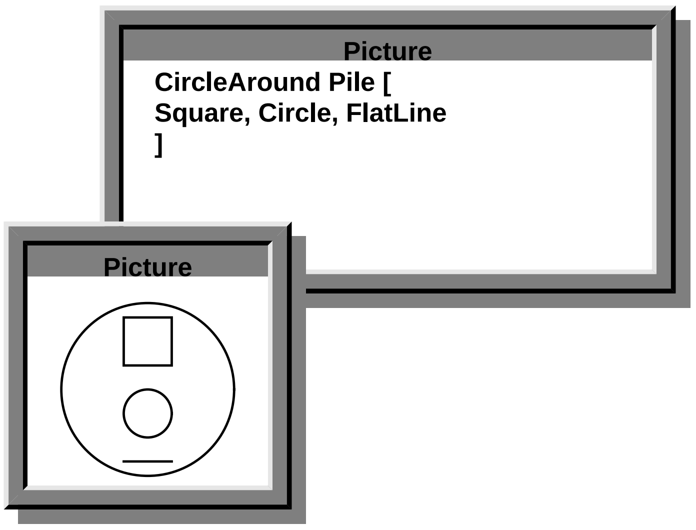

Figure 15 Two views of a picture object.

Just as each textual view of an object has an associated focus, so too does each graphics view of a picture object (from the systems point of view there is no difference between the two styles of view). The focus is indicated by high-lighting the elements contained in it and just as with text, the user is able to alter the scope of the focus with the appropriate commands or by interacting directly with the view using the mouse.

2.2 Incremental display of picture objects

It is possible to implement the display of picture objects so that they follow the incremental presentation methods described in Section 1.7. The one monolithic display invariant can then be broken down into many small invariants between individual picture elements and the parts of the view that they correspond to and, once again, the changing of a small part of a picture would result in an update of only the small area of the view that it was displayed in. Such an incremental display method would have to take account of overlapping elements.

3 Developing a graphics model

An underlying graphics model had to be developed that followed the general guidelines of the Views model. Having established this, the primitives and structures of the model had to be decided upon and how the picture structures should be presented in textual form.

What style of graphics should initially be supported by the model? The domain decided upon was schematic, two-dimensional pictures used to communicate information and relationships. This was decided upon after discussing what sort of diagrams were most frequently drawn, and after looking through various styles of literature at the types of diagrams used. Such a model is general enough to be of widespread use, yet simple enough to afford ease of use. It was also decided to offer support for less structured pictures through the eventual inclusion of bit-maps and pix-maps within the model.

3.1 Existing graphics models

The first task was the design of the hierarchical graphics structures necessary to describe pictures. The initial part of this task was an analysis of graphics models used in existing systems. Various systems were examined and evaluated in terms of how usable they were and how well they fitted in with the Views methodology [2]. In particular we looked at

Ideal [5], Pic [6], MacPaint [7], PostScript [8], GKS [9] and ILP [10] (see also [4]). The conclusions from this investigation were that two-dimensional graphics models could be divided up into the following categories:

- **Interactive paint systems:**
  For example MacPaint on the Apple Macintosh. These have good immediate feedback as to the changes the user is making to the picture. They have a simple structure (the bitmap) and complex operations to work on that structure (paint, draw, fill). The entire structure of the picture is apparent; there is no underlying complex structure. Thus lines, boxes and so on are all achieved through complex tools for manipulating the bitmap.
- **Interactive drawing systems:**
  These are similar to the above in that the user builds up a structure with the complex tools. However, the picture is usually more structured than with paint systems. Elements can be edited and manipulated once they have been defined. There is a strong relationship between the complexity of the structure and the complexity of the tools needed to define it.
- **Procedural graphics:**
  For example PostScript. These can be seen as an advanced paint and draw system that obeys textual commands. A file containing such a list of textual commands is thus a description of a two-dimensional picture. The feedback given by such systems is bad. The picture file must be processed in a batch manner before the result of changes can be seen and the structure must obey a strict and sometimes complicated syntax.
- **Semi-structured graphics:**
  For example PIC. The structure is still processed in a batch manner, but it is better structured and easier to use. Furthermore, the structure is still quite temporal in nature. Many parts of the specification depend heavily on what has already been specified.

Since the concepts underlying Views embody well structured objects operated on by a simple tool set we decided to work in the direction of a well structured non-procedural model.

4 The Views graphics model

The aims of the design of the primitive objects were the same as those for the rest of the Views system, they were to be simple, obvious and well unified. Achieving powerful and high-level effects while being controlled through low-level operations. The result was a collection of specifications that occurred in many other systems along with several new specifications and ideas that were new yet simple and powerful. The graphics specification language is included as an appendix and only the essential concepts are explained in the following description.

The user is able to use a variety of basic elements, for example line, square, rectangle, circle, curve. Pictures can be constructed by combining these elements with other elements using functions such as combine, row and pile. Pictures (and sub-pictures) can be transformed in various simple ways (repositioned, rotated and scaled). They can be enclosed within other elements (square around) and they can be given invisible margins (padded). Their appearance can be changed from that of plain solid black lines. Closed shapes (polygons, ellipses, circles) can be filled with colours or textures. There are also functions to specify lines and arrows between sub-pictures.

4.1 Coordinate and rotation schemes

Since the graphics is intended to be easily accessible to the user, and the majority of users are envisaged as not having a mathematical or computing background, the more conventional methods of expressing coordinates and rotation were chosen. The coordinates are expressed as positive and negative integers. The coordinate system used is the conventional ‘graph-paper model’; X positive to the right and Y positive upwards. Rotations are specified as a bearing; an integer indicating the rotation in a clockwise direction about the origin measured from the Y positive axis. Degrees were chosen as the units to measure the rotations in since radians were deemed as being unsuitable for everyday use. (If you ask someone how big a certain angle is they will very rarely give their answer in radians.) When specifying the bearings, negative values and values greater than 360 are accepted.

4.2 Text

Text can be included within a picture where it can be treated in just the same way as any of the other primitive graphics objects; it can be scaled, positioned, rotated, coloured and so on. Also, there is no inherent difference between the text that is included as part of a picture and the text that constitutes the paragraphs in a document. Both can be edited and manipulated in exactly the same way. The only difference being their context within another Views object.

4.3 Defaults

Many of the functions used require the specification of a distance of some sort. For example the edge length of a square or the separation of elements in a row operation. As well as this value being specified numerically there is also the operator ‘nice’ which can be used in place of a set distance. When nice is used the system chooses a distance that makes the resulting picture look well balanced. For example padding a picture with a distance of one tenth of its height (or its width if this is greater than the height).

If the distance is not specified then a distance of zero or the nice operator is assumed depending upon the function in question. In some cases the choice is obvious. A line specification without a distance will more likely be ‘line nice’ than ‘line 0’. Similarly for other picture elements. In other cases zero and nice could both be possible choices and the system must then use one of them as the standard default each time. For example row takes the nice operator as its default separation between elements.

5 Squiggol

Before moving on to give some examples we shall outline a few of the operators used in Squiggol, the language used to specify Views objects and invariants. Squiggol has several operators that apply to lists. One that we shall be using is the map operator (*) this takes a unary operator and applies it to each object in a list to yield another list comprising of the results of each application. Thus:

Double * [1, 2, 3]

yields:

[2, 4, 6]

Squiggol also allows operators with many operands to be converted to operators with less operands by ‘filling in’ one of the operands with a fixed value. For example the binary operator ‘Rectangle’ which returns a rectangle of the specified width and height can be used as a unary operator thus:

Rectangle (5, ♦)

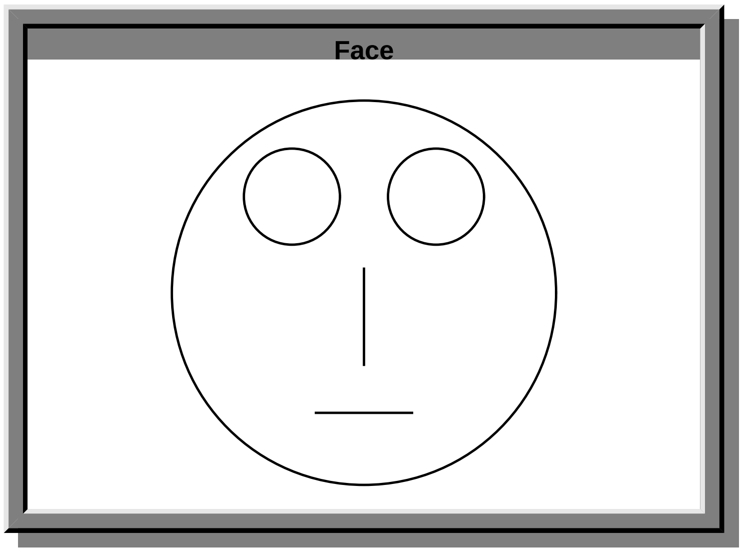

Figure 16 A simple picture of a face.

The ♦ symbol is a ‘hole’ which is filled in when the operator is applied to a single value to yield a rectangle of the appropriate height but with a fixed width of 5.

6 Examples

We shall now give three examples demonstrating how picture objects can be specified using the textual descriptions of the structure. The first of these is a simple picture of a face. It is composed of two circles and two lines all contained within one large circle. The surrounding circle can easily be implemented using the ‘CircleAround’ operation, thus the textual view of the face is shown below while the graphic view is shown in Figure 16.

```
Face =
CircleAround Pile[
FlatLine,
UprightLine,
Row[ Circle, Circle ]
]
```

The two circles for the eyes are grouped together using the ‘row’ operation and the resulting sub-picture is grouped together with the two lines (the nose and mouth) using the ‘pile’ operator. The initial padded’ operation gives the resulting picture an invisible padding so that when it is visited in a window it is displayed slightly clear of the windows edge.

In the face example the radii of the eyes are two objects of equal value residing in the picture structure. It is possible for the user to visit each of these objects and edit the value to produce an immediate change in the size of that particular eye. As has already been mentioned it is possible to use two instances of the same object for the eyes thus:

```
Eye= Circle 5
```

```
Face =
CircleAround Pile[
FlatLine,
UprightLine,
```

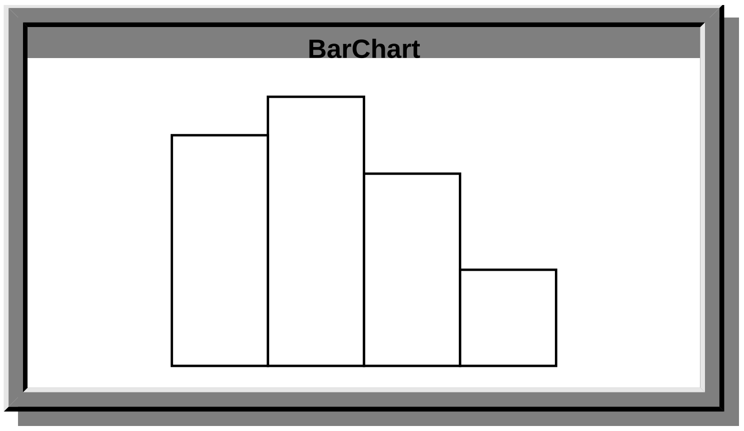

Figure 17 A bar-chart.

```
Row[ Eye, Eye]
]
```

Now if the user were to visit and edit the radius of the circle, both of the eyes would change size together.

6.1 A bar-chart

The second example demonstrates how certain aspects of a picture can depend upon other objects. We shall specify a simple bar-chart display of a list of values. The graphical view is shown in figure 17, the textual view is:

```
BarChart(NumberList)
= Row BotRectangle(5, ♦ ) * NumberList

TestList = [12, 14, 10, 5]
TestGraph = BarChart(TestList)
```

The first section sets up the function to display a list as a bar-chart. Our bar-chart is achieved by applying the ‘Row’ operator to a list of rectangles to yield a single picture composed of a row of rectangles. The rectangles are produced by applying a unary rectangle function to a list of values using the map operator. The second part of the bar-chart script applies this function to a list of numbers. Row aligns elements according to the position of their origin. Rectangle returns a rectangle with the origin at the centre. Thus ‘BotRectangle’ is defined to return a rectangle with its origin at the centre of the bottom edge:

```
BotRectangle(w, h)
= Rectangle(w, h) At (0, h/2)
```

6.2 A clock

Finally a more complex structure with an invariant link to the Views system clock. The Views system clock is a structured object composed of three numeric values representing hours, minutes and seconds. The values are updated every second with the effect that anything linked with an invariant to the seconds object, for example, will automatically be updated every second.

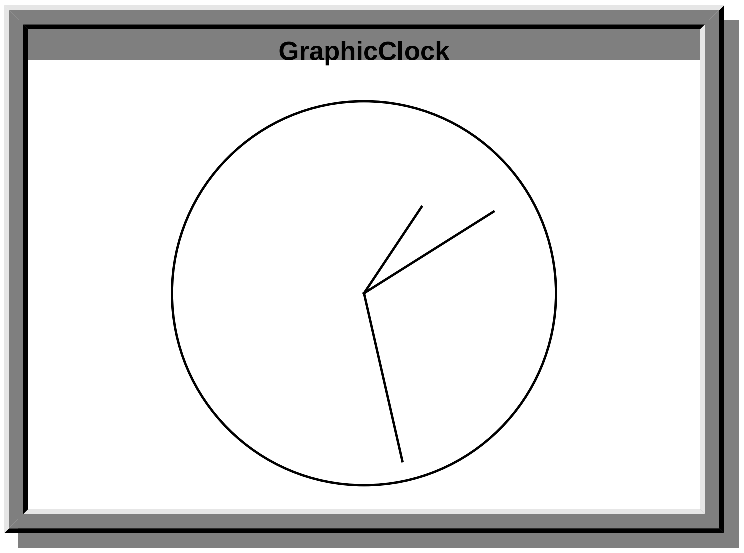

Figure 18 A graphic clock.

The basis of the clock picture are the three hands, the rotation of each will be dependent in some way upon the values of the hours, minutes and seconds objects such that, as the Views system clock changes, the hands on the graphic clock will change position in the expected manner. The text view is given below and the graphic view is shown in Figure 18.

```
GraphicClock(h, m, s) =
Combine[
Circle 4,
UprightLine 6 Rotated s * 6,
UprightLine 5 Rotated m * 6,
UprightLine 4 Rotated (h * 30 + m/2)
]
```

A picture object is constructed composed of the circle of the clock face combined with three hands, each of which is a line rotated by an amount dependent upon the values of the clock.

7 Future directions

7.1 Three-dimensions

The initial two-dimensional graphics model can easily be extended to three-dimensions to enable users to specify simple three-dimensional scenes. The extensions to each concept within the existing model are obvious: positions would have three coordinates, scaling and the other transformations would become more complex. New three-dimensional primitives would have to be included: spheres, boxes, cylinders and so on. Such a three-dimensional scene could be connected by a ‘geometric projection’ invariant to two other objects, a viewpoint object describing the parameters of a geometric view of the scene, and a two-dimensional picture object which would be the resulting two-dimensional projection of the scene using the given viewpoint.

7.2 Rendering and animation

As well as doing simple wire frame projections of three-dimensional scenes it would be possible to build Views based hidden-line removal applications or rendering engines that updated two-dimensional picture objects with more complex shading and texturing operators. Also, the breaking down of invariants into basic invariants makes Views a possible platform for animation. The three-dimensional projection described above could be broken down with the result that changes in the three-dimensional scene object would only cause partial update of the two-dimensional picture object. Thus if the three-dimensional scene were dynamic in nature the Views system would be able to display an animated view in a far more efficient manner than with conventional frame-based approaches to computer animation.

8 The basic functions

The basic functions have been divided up into three sections below; the primitives, the modifiers and the combining functions. The notation used here is that those parts written in bold face type are the literal parts of the specification (the bits that don’t change, the fixed frame work of the specification) and those parts in italics are the parameters (the bits that the user ‘fills in’ according to what effect is wanted).

8.1 Primitives

The primitives that make up the Views graphic structures are listed below.

Lines

There are several ways of specifying lines:

- UprightLine length
  A vertical line of a given length. Omitting the length specification yields a line whose length is dependent upon the context.
- FlatLine length
  Similar to above. A horizontal line of a given length.
- Line(pos1, pos2 ... posN)
  A line connecting two or more points.
- Polygon(pos1, pos2... posN)
  A polygon created from the list of points.

Arrows

Arrows are specified in a similar way to lines:

- ArrowUp length
  An arrow pointing upwards of length Length. (similarly for down, left and right.)
- Arrow(pos1, pos2 ... posN)
  An arrow running through two or more points.

Other simple elements

The remaining simple elements are dots, text, arc and curves:

- Dot
  A small visible dot.
- Text(exp)
  The expression is displayed as a piece of text.

- Arc(start, end)
  A circular arc centred at the origin, drawn in a clockwise direction, starting at point ‘start’ and finishing where it cuts the line joining the centre to ‘end’.
- Arc(radius, startbearing, endbearing)
  Alternatively an arc can be specified in terms of a radius and two bearings delimitating the extent of the arc. Once again the arc is drawn in a clockwise direction.
- Curve(pos1, pos2... posN)
  A smooth open curve joining the positions in the list.
- Loop(pos1, pos2... posN)
  A closed curve joining the points in the list. (The area-enclosing equivalent to curve.)

Area-enclosing elements

As well as polygon and loop there are also a number of other area-enclosing elements. Area-enclosing elements can be treated as non area-enclosing and just drawn as lines with different styles, or they can be explicitly treated as area-enclosing and drawn as filled areas in different fill styles:

- Circle radius
  A circle with the given radius.
- Ellipse w h
  An ellipse oriented with one axis parallel to the X axis, with its centre at the origin and with the given width and height.
- Square e
  A square, oriented orthogonal to the X and Y axes, with its centre at the origin and with edges of the given length.
- Rectangle w h
  A rectangle, oriented orthogonal to the X and Y axes, centred at the origin and with the given width and height.

Furthermore area-enclosing elements can be yielded by combining non-area-enclosing elements, (the combine operation is described later). For example:

```
Combine[ Arc(5, 0, 180), UprightLine 10 ]
```

8.2 Modifiers

A picture made up from one or more of the above can be enhanced by one of the following:

Picture transformations

These apply some of the more common two-dimensional transformations to a picture:

- pic At pos
  Reposition the origin of the picture according to the position given.
- pic Rotated angle
  Rotate the picture by the given angle (in degrees) clockwise about its origin.
- Mirrored pic
  Create a mirror image of a picture, left to right about the vertical axis. (an up-down mirrored image, should it be required, can be achieved using left to right mirroring and rotate).

- pic Scaled scale-factor
  The scale factor can be either two parameters; meaning scale the picture in the X and Y directions using the two scale factors, or it can be simply one factor meaning scale the picture in both the X and Y directions by the same factor.

Picture properties

A picture can have other properties besides its shape. The shape itself can be drawn in different manners. The default is solid black lines of a standard width but things can be changed by use of the operations in the list below. When properties are assigned to a picture object they take effect on that object and propagate ‘down’, in a recursive manner, through all the sub-objects. This propagation of properties can be over-ridden by another property specification further ‘down’ within the sub-objects. Thus in:

```
Blue  CircleAround (
Row[ UprightLine, Red Square]
)
```

the ‘Blue’ effects the surrounding circle and the line, but for the square the ‘Red’ overrides the ‘Blue’.

- mode Drawn pic
  The picture is drawn in a line style specified by Mode, (e.g. dotted lines, dashed lines, invisible).
- Red pic
  The picture is drawn in red (similar commands for other colours).
- pattern Filled pic
  If the picture is area-enclosing then the area enclosed is filled with the pattern specified, of which there are a number of standard types.

Picture related values

These are operations applied to pictures which yield a position related to the picture. The algorithms for deciding the positions are designed to yield ‘sensible’ results.

- Centre pic
  The centre of Pic.
- distance NorthOf pic
  The position a given distance from a picture in a Northerly direction, (similarly for all eight major points of the compass; NorthOf, NorthWestOf, WestOf and so on).
- Width pic
  This returns the width of a picture, (a similar command returns the height).

Complex elements

A number of operations modify pictures or combine more than one picture to yield a new picture:

- Line(pic1, pic2)
  The line drawing command we encountered earlier can be used with one or both position arguments replaced by pictures. The result of the above example is a line joining one picture to the other.
- Arrow(pic1, pic2)
  Similar to above but with an arrow.

- SquareAround pic
  The picture is drawn centred within the smallest square that will enclose it. As well as SquareAround the system supports; RectangleAround, CircleAround, EllipseAround And HullAround. When HullAround is used the result is a simple polygonal hull enclosing and following the boundaries of the picture.
- margin Padded pic
  The picture is padded by an amount of whitespace given by the value of ‘margin’. Such a modification will only of course become apparent when it is subject to some modification that depends upon the picture boundaries such as RectangleAround or inclusion in a Row.
- gap Pile(pic1, pic2... picN)
  The pictures are positioned above one another, separated by a distance given by Gap.
- gap Hang(pic1, pic2... picN)
  The pictures are positioned below one another, separated by a distance given by Gap.
- gap Row(pic1, pic2... picN)
  The pictures are positioned beside one another, from left to right separated by a distance given by Gap.
- Combine(pic1, pic2... picN)
  The pictures are drawn overlapping each other with their origins at the same point. (If any of the pictures have filled elements, those at the beginning of the list will obscure those occurring later in the list.)

9 Bibliography

[1] L.J.M. Geurts, S. Pemberton and L.G.L.T. Meertens. *The ABC Programmer's Handbook.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[2] L. Meertens and S. Pemberton. *The Ergonomics of Computer Interfaces - Designing a System for Human Use.* CWI Report CS-R9258, December 1992, CWI Amsterdam.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[4] Jaap Zwaan and Rolf Zwart. *Graphics for ABC*. CWI Report CS-R9255, December 1992, CWI Amsterdam.

[5] C.J. van Wyk. *IDEAL Users Manual.* Bell Laboratories, Murray Hill, New Jersey, 1979.

[6] B.W. Kernighan. *PIC A language for typesetting graphics.* Bell Laboratories, Murray Hill, New Jersey, 1982.

[7] Carol Kaehler. *MacPaint*. Apple, Cupertino, California, 1983.

[8] Adobe Systems. *PostScript Language Tutorial and Cookbook.* Addision Wesely, Wokingham England, 1985.

[9] F.R.A. Hopgood, D.A. Duce, J.R. Gallop and D.C. Sutcliffe. *Introduction to the Graphic Kernel System GKS*. Academic Press, 1983.

[10] P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint and A.H. Veen. *Intermediate Language for Pictures.* Mathematical Centre Tracts 130, ISBN 90 6196 2048, Amsterdam, Netherlands, 1980.