

# TEXT, IMAGE, GRAPHICS EDITOR

## A THESIS

Submitted To The Department Of Computer  
Engineering And  
Information Sciences  
And The Institute Of Engineering And Sciences  
Of Bilkent University  
In Partial Fulfillment Of The Requirements  
For The Degree Of  
Master Of Science

By

Ahmet Coşar  
September 1988

THESIS  
QA  
76-6  
C82  
1988

# TEXT, IMAGE , GRAPHICS EDITOR

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER  
ENGINEERING AND  
INFORMATION SCIENCES  
AND THE INSTITUTE OF ENGINEERING AND SCIENCES  
OF BILKENT UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE

By

Ahmet Coşar

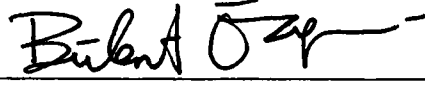
September 1988

*Ahmet Coşar*

tarafından başlanmıştır.

QA  
76.6  
C82  
1988  
B1872

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



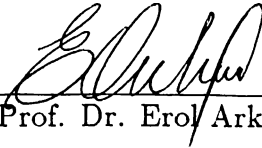
Assoc. Prof. Dr. Bülent Özgüç (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Dr. Mehmet Baray

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Dr. Erol Arkun

Approved for the Institute of Engineering and Sciences:



Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Sciences

# ABSTRACT

## TEXT, IMAGE , GRAPHICS EDITOR

Ahmet Coşar

M.S. in Computer Engineering and  
Information Sciences

Supervisor: Assoc. Prof. Dr. Bülent ÖZGÜÇ

September 1988

The editor proposed in this study can manipulate textual, graphics and image data in a unified way. Each data type can be edited individually or dependencies can be set up between various data items so that modifying one might propagate its effects on others. The system is developed by using new software tools and techniques such as object oriented programming, multi window workstations running with event selection principles and iconic interfacing. Facilities for data protection, such as journaling are provided. Data storage and editing principles are handled within guidelines of well established standards. However, where such definitions fall short, proposals for new techniques are made especially with respect to relation sets binding various data types.

Keywords : Text, image, graphics, user interface, window manager, object-oriented programming.

# ÖZET

## YAZI, RESİM, ÇİZİM İŞLEMCİSİ

Ahmet Coşar

Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans

Tez Yöneticisi: Doç. Dr. Bülent ÖZGÜÇ

Eylül 1988

Bu çalışmanın sonucunda geliştirilen sistem yazı, çizim ve görüntü verilerini tek tek ya da aralarında ilişkiler tanımlayarak bir bütün olarak işleyebilmektedir. Bu yöntemle herhangi bir veride yapılan bir değişiklik diğer verileri de etkileyebilmektedir. Sistemin geliştirilmesinde birçok yeni yazılım geliştirme teknikleri ve hazır yazılımlar kullanılmıştır. Bunlardan başlıcaları, nesnel yaklaşımli programlama, çok pencerele iş istasyonları, imgesel kullanıcı arayüzeyidir. Veri saklama ve koruma amacıyla yedekleme olanakları da sağlanmıştır. Bu yazılım geliştirilirken genel olarak kabul görmüş standartlara uyulmaya çalışılmış ancak gerekli olduğu zaman, özellikle değişik veri türleri arasında ilişki tanımlama amacıyla, bir takım yeni teknikler kullanılmıştır.

Anahtar Kelimeler: Yazı, çizim, görüntü, nesneye-yönelik programlama, etkileşim sistemleri, çok pencerele iş düzeni.

## ACKNOWLEDGEMENT

I would like to thank my thesis advisor, Assoc. Prof. Bülent ÖZGÜÇ for his guidance and support during the development of this study.

I also appreciate my colleagues, Mesut Göktepe and Aydın Kaya for their valuable discussions and comments.

# TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2</b>	<b>STANDARDS AND RELATIONS ON DATA</b>	<b>5</b>
2.1	STANDARDS ON DATA . . . . .	5
2.1.1	Graphics data standards . . . . .	5
2.1.2	Image data standards . . . . .	12
2.1.3	Text data standards . . . . .	14
2.2	DOCUMENT ARCHITECTURE MODEL . . . . .	17
2.2.1	Document structures and objects . . . . .	17
2.2.2	Content portions . . . . .	18
2.2.3	Object types and their characteristics . . . . .	18
2.2.4	Object classes and object definitions . . . . .	22
2.2.5	Document classes and document definitions . . . . .	23
2.2.6	Overall document architecture model and document profile . . . . .	23



<b>3</b>	<b>RELATIONS BETWEEN DATA TYPES AND DATA MANIPULATION</b>	<b>25</b>
3.1	RELATIONS BETWEEN DATA TYPES . . . . .	25
3.2	DATA MANIPULATION . . . . .	26
3.3	DATA STORAGE . . . . .	29
3.4	DATA STRUCTURES . . . . .	30
<b>4</b>	<b>SESSION CAPTURE, ARCHIVING AND BACKUP, UNDO, REDO</b>	<b>35</b>
4.1	UNDO AND REDO IMPLEMENTATION	37
<b>5</b>	<b>USER INTERFACE COMPONENTS</b>	<b>39</b>
5.1	WINDOWs . . . . .	39
5.2	MENUs . . . . .	40
5.3	PANELs . . . . .	41
5.4	INPUTS, EVENTS, SELECTION . . . . .	41
5.5	INTERRUPTS AND THEIR MANAGEMENT . . . . .	43
<b>6</b>	<b>CONCLUSIONS</b>	<b>47</b>
<b>A</b>	<b>OPERATIONS THAT ARE CURRENTLY AVAILABLE IN THE SYSTEM</b>	<b>49</b>
A.1	TEXT OPERATIONS . . . . .	50
A.1.1	Moving text items . . . . .	50

A.1.2	Resizing text items . . . . .	50
A.1.3	Editing text data . . . . .	51
A.1.4	Defining text items . . . . .	51
A.1.5	Selecting text items . . . . .	51
A.2	IMAGE OPERATIONS . . . . .	52
A.2.1	Moving image items . . . . .	52
A.2.2	Resizing image items . . . . .	52
A.2.3	Editing image items . . . . .	52
A.2.4	Defining image items . . . . .	52
A.2.5	Selecting image items . . . . .	53
A.3	GRAPHICS OPERATIONS . . . . .	53
A.3.1	Moving graphics items . . . . .	53
A.3.2	Resizing graphics items . . . . .	53
A.3.3	Editing graphics items . . . . .	54
A.3.4	Defining graphics items . . . . .	54
A.3.5	Selecting graphics items . . . . .	54
<b>B</b>	<b>The Language of The System</b>	<b>56</b>

# LIST OF FIGURES

1.1	A sample document having relations among text and image data. . . . .	2
1.2	A rectangle with its defining attributes. . . . .	2
2.1	Graphics Standards at Different Levels of Data. . . . .	7
2.2	Logical object structure. . . . .	19
2.3	Layout structure. . . . .	20
2.4	ODA document architecture. . . . .	24
3.1	The "circle" and the circle with the attribute definitions of the circle. Circle_X and Circle_Y give the center of the circle. . .	26
3.2	Positioning text data inside a rectangle. . . . .	27
3.3	Changing font size as covering rectangle shrinks and expands. . . . .	27
3.4	Placing text data inside graphical items. . . . .	29
3.5	The object structure. . . . .	30
3.6	The attribute structure. . . . .	31
3.7	The page structure. . . . .	31

3.8	A sample two page document. . . . .	32
3.9	Attributes of a rectangle. . . . .	33
3.10	Attributes of a circle. . . . .	33
3.11	Attributes of an ellipse. . . . .	33
3.12	Attributes of an image. . . . .	34
4.1	A sample file saved on disk. . . . .	36
5.1	A general layout of the screen. . . . .	40
5.2	Notifier–Selection mechanism. . . . .	42
5.3	Notifier–Selection mechanism with broadcast messages.	44
5.4	Broadcast algorithm for performing necessary updates. . . . .	46
A.1	The result of moving graphics items to the same point. . . . .	53
A.2	The effects of resizing graphics items. . . . .	54
A.3	Editing operations on polygons. . . . .	55

# 1. INTRODUCTION

The purpose of this work is to define an environment for the manipulation of three different data types namely text, image and graphics. There are many products that manipulate text, image or graphics data independently, but most of these systems are unable to manipulate them in a unified document. The problem is even more complex when the user has documents containing relations between different data types such as relative positions and lengths.

There are two possible approaches in providing such relation definitions. The first one is a set of predefined operations that can be selected through menus, function keys, keywords, etc. The limitations of such a system are obvious but the implementation is easier and more efficient code can be written for each relation. In the second approach each data item has attributes associated with it and these attributes are used while processing that data item for display. These attributes can reference other data items through the usage of some expressions that can be edited by the user. The data items of such a system are dynamic since expression evaluations are performed for other possible changes when a particular item is modified. In this environment it is possible to define documents with relations such that changing a text item may cause a graphical or image item to be modified if the text attributes are accordingly prepared (Figure 1.1).

For each item these attributes are stored in a linked list and users are free to add or update them using either the keyboard or the mouse. Figure 1.2

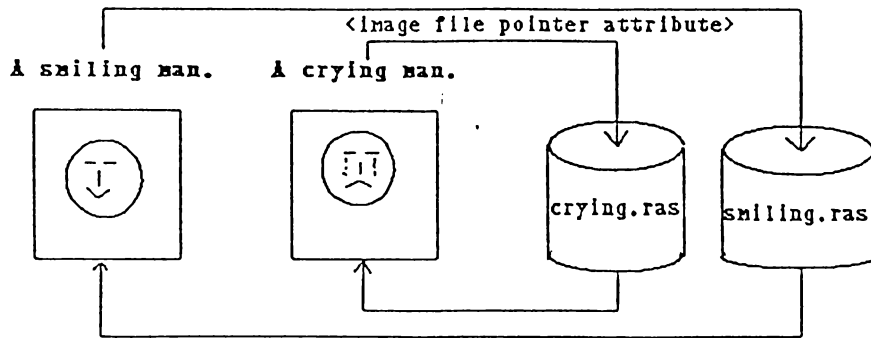


Figure 1.1: A sample document having relations among text and image data.

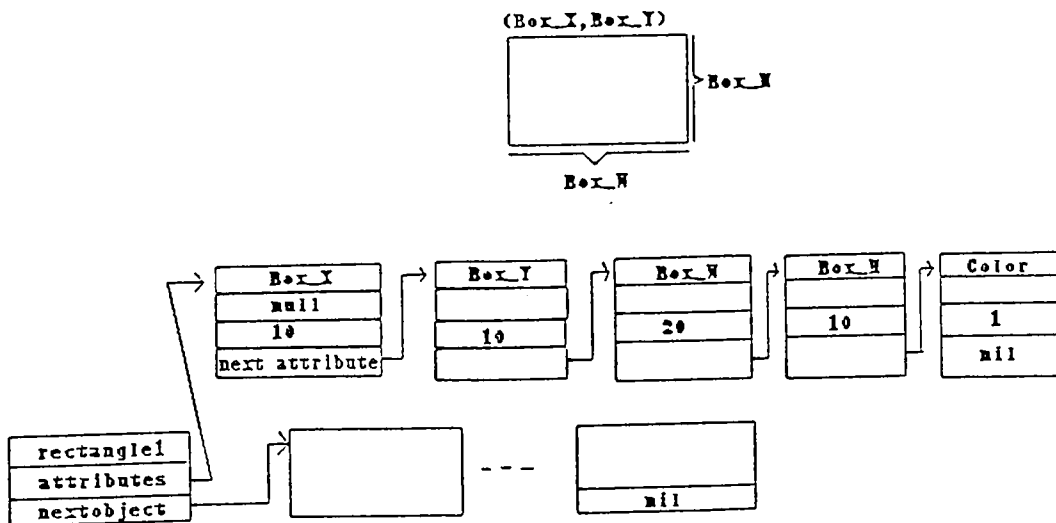


Figure 1.2: A rectangle with its defining attributes.

shows a graphical object with its associated attributes and the data structures used for storing these attributes. For each data type (i.e. text, image and graphics) there is a special display procedure (e.g. `Graphics_Display()`) that is called with the item to be displayed as its argument. This procedure checks the item attributes while displaying it on the screen. This type of an implementation is general and modular in nature and the addition of a new attribute is greatly simplified. A simple example of setting color is discussed next to clarify the effects of the attributes. An attribute with the name *Color* and an integer value between 0 and 255 acting as an index to a colormap table are to be used. The C code that concatenates this information to the display parameter is given below. The attribute structure is as defined above and the pixel operation specifies a logical operation (like OR or XOR) among the corresponding bits of the image to be displayed and the image that already exists in the frame buffer of the workstation. `PIX_COLOR` [16] is a macro that is used for inserting color information into the pixel operation.

```

if(A= (struct attrib *)exist_attr(item,"Color"))
    pixel_operation |= PIX_COLOR(A->attrval);

```

It is possible to define each attribute either by an integer value or a string expression. In order to allow users to define their relations using this simple language special primitives have been included. The use of such a language has made our system different than the ODA (Office Document Architecture)[7] standard since it is not possible to represent all of the relations that can be defined by such a language using the relations provided by ODA. This, however, does not mean that one cannot represent the same document by using ODA. A simple relation is given below with an expression used for defining two items to have the same color.

```

Color= 'otheritemname 'Color \ival

```

The quotation marks preceding words denote that those words are not evaluated and pushed into a stack and are later used by other commands as parameters. The backslash character precedes the commands of the language (as in the TeX convention [13]). The `\ival` command that stands for item value pops two strings from stack. The first popped string is the attribute and the second one is the item name that is unique for each item. The attributes are then evaluated recursively. Since this may cause infinite loops with a cyclic relation, special checks are made to terminate the recursion when an item is revisited. Chapter 5 explains this effect in detail.

By setting various relations between different data items of any type, users will not have to deal with details that they have already defined. The implication of this is when a change in only one data item is made, all other data items might be updated accordingly if this is specified in the relation set. One such relationship is defined for inserting text into a text data that causes the rest of the line to be shifted right and if the right justification function is already defined for that text data, all of the text will be realigned without any additional user command. This function is already used in commercial text editors (e.g. Wordstar<sup>1</sup>). Similar examples can be given for image and graphics data as well.

---

<sup>1</sup>Wordstar ©, MicroPro International Corp.



## 2. STANDARDS AND RELATIONS ON DATA

### 2.1 STANDARDS ON DATA

Since the main objective of this work is to define standard operations for defining relations between text, image, and graphics data elements, we must have the standard definitions of these data types as well. Otherwise, defining a standard above non-standard objects cannot give much benefit to us.

#### 2.1.1 Graphics data standards

A graphics element is a drawing possibly including lines, geometric shapes (circles, ellipses), or some graphics objects already defined in the graphics database. Several operations on graphics data are supplied, some of them are:

- graphical transformations
- painting closed polygons
- clipping

Unfortunately the hardware devices used for computer graphics operations are not always compatible with each other both in the data structures they use and in the ways they manipulate these data structures. This fact

causes the users to introduce higher level interfaces to be able to do device independent graphics applications.

National Computer Graphics Association (NCGA) has endorsed the adoption and widespread use of the Graphical Kernel System(GKS) as the first family of compatible standards for computer graphics [9,6].

Several standards are currently in effect for computer graphics:

IGES(Initial Graphics Exchange Standards) [8] and NAPLPS (North American Presentation-Level Protocol Syntax) [12] are two examples for these standards. IGES provides a standard file format for transporting CAD/CAM design data between different systems. NAPLPS is a compact code for encoding the graphical and textual content of a picture for transmission and storage.

GKS is close to attaining official status. Balloting at the ISO (International Standards Organization) is closed. In the US, GKS was in the public review stage of the ANSI procedure, in 1985.

The ANSI public review stage of VDM (Virtual Device Metafile) [18] closed May 6, 1984, and the corresponding stage at the international level was put into progress. VDM is intended to be a graphics picture file standard concerned with the transfer of sufficient device independent information to enable a picture to be regenerated on a wide range of graphics devices.

VDI (Virtual Device Interface) [1] is behind VDM. The first letter ballot had begun in April 1984. VDI is a two way communication protocol that takes place at the lowest level of device independence.

The PHIGS (Programmer's Hierarchical Interactive Graphics System) [3] is in the early stages of development.

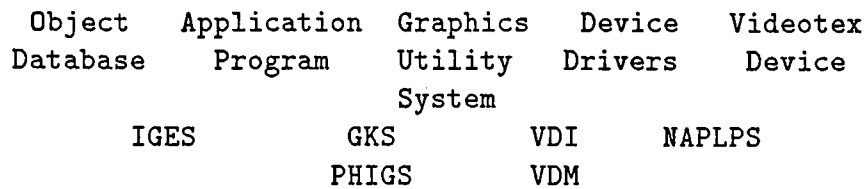


Figure 2.1: Graphics Standards at Different Levels of Data.

From a general view, all of these efforts attempt to standardize interfaces. IGES operates at the level between the object database and the application program. GKS and PHIGS interface between the application program and the graphics utility system. VDI and VDM are positioned between the graphics utility system and device drivers, and NAPLPS functions between a NAPLPS device driver and a videotex device, as shown in Figure 2.1.

CORE [19] is a graphics system for creation, modification, and manipulation of three-dimensional, planar faced objects. Every object in a scene consists of one or more planar polygon faces. A face may be defined by supplying its vertices as points in space, or alternatively by naming points already in the (partial) description of the object. In order to *create* or *modify* an object description, the object must be *opened*. When done it is closed, and it may be re-opened later for modification.

Colors are associated with objects and individual faces by specifying a color table index. Faces can have different colors on each side, including *invisible*. Object can be translated, rotated, and scaled. The *camera* is also an object and can be similarly manipulated. The camera *focal length* can be specified to produce varying magnification of the scene. See table 1.

The Graphical Kernel System (GKS) is the first international standard for computer graphics programming. CORE was developed before GKS but it is not as widely accepted as GKS due to the complexities of programming

TABLE 1

CORE	
CREATION COMMANDS	DELETION COMMANDS
CR-OBJECT NAME COLOR CR-FACE POINTLIST [NAME [CLR1 [CLR2]]] CR-LINE X1 Y1 Z1 X2 Y2 Z2 NAME COLOR CR-AXIS X1 Y1 Z1 X2 Y2 Z2 NAME CR-POINT X Y Z [NAME]	DELETE-OBJECT {OBJECT}... DELETE-LINE LINENAME DELETE-POINT X Y Z DELETE-FACE FACENAME
DUPLICATION COMMANDS	MODIFICATION COMMANDS
COPY-OBJECT OBJECT NEWNAME	RENAME-OBJECT OBJECT NEWNAME COLOR-OBJECT OBJECT CCW-CLR CW-CLR PLCOLOR FACE CCW-COLOR CW-COLOR OPTIMIZE OBJECT FUZZ
COMBINE COMMANDS	COLOR COMMANDS
MERGE INTOOBJECT OB1 .. OBN CONCATENATE INTOBJECT OB1 .. OBN	COLORTABLE N COLORLIST GETCOLOR IND1 .. INDN CHANGECOLOR INDEX COLORLIST SET BACKGROUND N
MOVEMENT COMMANDS	VIEWPOINT COMMANDS
TRANSLATE OBJECT DX DY DZ MOVE OBJECT POINT X Y Z SCALE OBJECT XS YS ZS XC YC ZC ROTATE OBJECT THETA XT YT ZT XH YH ZH ROTATEX OBJECT THETA ROTATEY OBJECT THETA ROTATEZ OBJECT THETA ROTATEV OBJECT THETA X Y Z	SET-VIEW X1 Y1 Z1 X2 Y2 Z2 SET-WINDOW HX HY SET-ANGLE ANGLE SET-VIEWPORT LLX LLY URX URY CENTER-DISPLAY

by CORE. GKS defines an interface for programming device-independent graphics applications. The graphics model includes concepts such as segments, logical input devices, and workstations. GKS was developed as a two dimensional graphics system, and now a three-dimensional extension is being considered. Even though GKS was selected as the standard for doing graphics in a device independent environment, its functions are mainly for communicating with different graphics devices. This is the reason why it was initially designed as a two dimensional system. Even if the three-dimensional extension is successfully done, problems will ( and do) arise in areas where efficient manipulation of graphics data is important. See table 2.

To address this problem a new standard, PHIGS (Programmer's Hierarchical Interactive Graphics System), has been proposed. PHIGS is compatible with GKS and it was designed on top of GKS by adding flexibility in data structures and defining additional functions for dynamic manipulation of graphics data.

PHIGS is a graphics standard being developed under the regulations of the ANSI.

PHIGS Structures: In PHIGS the whole graphical data is seen as a tree of *structures* and each structure is a list of *structure elements* that can be:

- graphical primitives (line, marker, text, polygon, etc.)
- an attribute or view selection
- a transformation matrix
- an *Execute Structure* element

In PHIGS, structures can be edited dynamically in run time without re-definition (which is not possible in GKS), and the attributes are bound to

TABLE 2

GRAPHICAL KERNEL SYSTEM(GKS)		
LEVELs		STATEs
INPUT	OUTPUT	
a- NO INPUT b- REQUEST INPUT ONLY c- FULL INPUT	m- MINIMAL 0- ALL PRIMITIVES AND ATTRIBUTES 1- BASIC SEGMENTATION WITH FULL OUTPUT 2- WORKSTATION INDEPENDENT SEGMENT STORAGE	GKCL- GKS CLOSED GKOP- GKS OPEN WSOP- WORKSTATION OPEN WSAC- WORKSTATION ACTIVE SGOP- A SEGMENT OPEN
DATA TYPES	CONTROL FUNCTIONS	OUTPUT FUNCTIONS
I- Integer R- Real S- String P- Point(x,y) . WC- World Coordinates . NDC- Norm.Dev.Coörd. . DC-Device Coordinates N- Name, an identifier E- Enumeration F- File W- Workstation type C- Connection id. D- Data Record	OPEN GKS CLOSE GKS OPEN WORKSTATION CLOSE WORKSTATION ACTIVATE WORKSTATION DEACTIVATE WORKSTATION UPDATE WORKSTATION REDRAW All Segments UPDATE WORKSTATION SET DEFERRAL STATE MESSAGE ESCAPE	POLYLINE POLYMARKER TEXT FILL AREA CELL ARRAY GENERALIZED DRAWING PRIMITIVES . Circular Arc . Circular Sector . Spline Curve
TRANSFORMATION FUNCTIONS		
NORMALIZED TRANSFORMATIONS		WORKSTATION TRANSFORMATIONS
SET WINDOW SET VIEWPORT SET VIEWPORT INPUT PRIORITY SELECT NORMALIZATION TRANSFORMATION SET CLIPPING INDICATOR		SET WORKSTATION WINDOW SET WORKSTATION VIEWPORT
INPUT FUNCTIONS		
INITIALIZATION OF INPUT DEVICES	SETTING OF INPUT DEVICES	REQUEST INPUT FUNCTIONS
INITIALIZE LOCATOR INITIALIZE STROKE INITIALIZE VALUATOR INITIALIZE CHOICE INITIALIZE PICK INITIALIZE STRING	SET STROKE MODE SET VALUATOR MODE SET CHOICE MODE SET PICK MODE SET STRING MODE	REQUEST LOCATOR REQUEST STROKE REQUEST VALUATOR REQUEST CHOICE REQUEST PICK REQUEST STRING

TABLE 2 - continued

SEGMENT FUNCTIONS	
SEGMENT MANIPULATION FUNCTIONS	SEGMENT ATTRIBUTES
CREATE SEGMENT CLOSE SEGMENT RENAME SEGMENT DELETE SEGMENT DELETE SEGMENT FROM WS ASSOCIATE SEGMENT TO WS INSERT SEGMENT	SET SEGMENT TRANSFORMATION SET VISIBILITY SET HIGHLIGHTING SET SEGMENT PRIORITY SET DETECTABILITY
OUTPUT ATTRIBUTES	
WORKSTATION INDEPENDENT PRIMITIVE ATTRIBUTES	WORKSTATION ATTRIBUTES
SET POLYLINE INDEX SET LINETYPE . solid . dashed . dotted . dashed-dotted SET LINEWIDTH SCALE FACTOR SET POLYLINE COLOR INDEX SET MARKER TYPE SET MARKER SIZE SCALE FACTOR SET POLYMARKER COLOR INDEX SET TEXT INDEX SET TEXT FONT & PRECISION SET CHARACTER EXPANSION FACTOR SET CHARACTER SPACING SET TEXT COLOR INDEX SET CHARACTER HEIGHT SET CHARACTER UP VECTOR SET TEXT PATH SET TEXT ALIGNMENT: left,right SET FILL-AREA INDEX SET FILL AREA INTERIOR STYLE SET FILL AREA STYLE INDEX SET FILL AREA COLOR INDEX SET PATTERN SIZE SET PATTERN REFERENCE POINT SET ASPECT SOURCE FLAGS	SET POLYLINE REPRESENTATIONS SET POLYMARKER REPRESENTATIONS SET TEXT REPRESENTATIONS SET FILL-AREA REPRESENTATIONS SET PATTERN REPRESENTATIONS SET COLOR REPRESENTATIONS
METAFILE FUNCTIONS	
WRITE ITEM TO GKSM GET ITEM FROM GKSM READ ITEM FROM GKSM INTERPRET ITEM	

structures while they are being traversed for display, that is called by *Execute Structure* element. Since our design includes functions for a user interface for doing editing on graphics data, PHIGS is used as the basis of implementing editing functions on graphics data. In fact, PHIGS includes functions for text editing as well, thus only image data must be handled apart from PHIGS.

Viewing Transformations: PHIGS allows the users to define their data in Modelling Coordinate (MC) system, and then maps them to an image in Device Coordinate (DC) system while displaying. In fact the mapping includes three intermediate coordinate system transformations:

- modelling coordinate system
- world coordinate system
- viewing coordinate system
- normalized projection coordinate system
- device coordinate system

### 2.1.2 Image data standards

Images can be obtained from many sources using different technologies. For example, video cameras convert light reflected from objects into an electronic signal that can be easily digitized. It is also possible for such devices to produce color information. They can also control the direction and resolution of sensors and use their own light sources as well. Smoothing and enhancing of images are other functions that can be performed by such devices.

While generating images we can also include information such as orientation of the object surface, velocity, or range of the object. Such information can be obtained by processing the image later.



In order to represent the abstraction of an image, image functions are utilized. Usually, an image function is a vector valued function of a vector parameter. The digital (discrete) image function is a special image function where all of the elements of the parameter are integers. Different image functions can be defined for the same image depending on the characteristics of the image that we want to represent. An example of such a function is  $f(X)=f(x,y)$  where the value gives the gray level intensity of the image in coordinates  $(x,y)$ . A color image can be represented by a vector valued function with three components each one giving the brightness of the image in red, green, and blue. All of the RGB monitors need such an image for display. Another problem that must be solved is to convert the continuous image function into a discrete function preserving the quality of the image. The number of gray levels that can be used is also limited and for a one bit plane monitor there are only two levels that are black and white. To display gray scale pictures on such devices, techniques such as dithering must be employed.

Today there exist many graphics workstations that have a resolution of 1000x1000 or higher and each pixel can have a color value chosen among thousands of available colors. However, representing each pixel on the screen with the absolute color values is not practical due to the large amount of memory required. Instead, each pixel is represented by an index (usually just eight bits long) to a table of 256 (if we use 8-bits long index) elements where each row consists of a set of red, green, and blue density values to represent the color of that pixel. This table is called the colormap-table and is useful for representing a high number of available colors by using a small number of bits per pixel. This technique is used for encoding images as well, since it reduces the amount of storage required fundamentally and can provide high quality images if the number of bits per pixel and the colormap table to be used are determined with care.

In this system images are saved in rasterfile [16] format. The first three parameters of this form define the width, height and depth (in bits) of the image. Depth of an image is determined by the number of different colors used in the image and these colors are assigned to single pixels as an index to the colormap table of that image. For example, a monochrome image has only two colors in it and one bit is enough to address these colors in the colormap table, assuming that these colors are placed at the beginning of the colormap table. Since the maximum number of entries in the colormap table in our application is 256, the maximum depth is eight while displaying an image. But it is possible to store and manipulate images that have a larger depth. These three parameters are followed by the colormap table. The remaining part of the file is the image to be interpreted according to the above defined header information.

### **2.1.3 Text data standards**

Text elements are arranged in a hierarchy such that text data consists of paragraphs that in turn consist of lines, and finally lines consist of characters. While moving down this hierarchy each paragraph, line, or character can have its attributes changed to affect the text data under lower hierarchies. The users are able to define their own hierarchies by defining a nesting operation on the text data. Each attribute setting will be saved while entering a new hierarchy and restored upon exit from the hierarchy.

The attributes that can be changed include fonts, line and paragraph styles, line lengths, spacings between lines and characters, etc. A color attribute selection is also available for displaying text data in different colors. In order to allow multiple text styles, available text formatters, like nroff and ditroff [10] are used. However, some of the widely used functions are included

in the system (e.g. right and left justification of text data) for efficiency purposes.

**Fonts:** It is necessary to use characters of different fonts in printed text for making some points clearer, and also to convey a special meaning to the reader. For example, in a manual, text written in bold letters may be those entered by the user from keyboard and the output on the screen might be printed in italic so that users can differentiate the responses of the system from user commands.

Many printers and computers come along with a number of fonts either in hardware or in software libraries. Some of the commonly used fonts are:

- times roman
- times italic
- times bold
- typewriter, etc.

There are also special fonts for mathematics and letters from foreign languages (other than English). The users can define their own fonts as well, but this is a very tedious work and takes very long time. What users usually want to do is to change the sizes of characters from these already defined fonts. Such an operation is very complicated and requires the fonts to be defined in such a way that they can be freely expanded and shrunk that is obviously not possible with a simple bitmap. A simple solution to this problem is to redefine the same font for each different size, that is of course a space consuming solution. The first method is chosen for our system and the fonts are defined in an appropriate structure.

Another problem that may occur while printing image or graphics data is the difference in the resolutions of screens and printers. This problem

is especially predominant with image data since it is not easy to perform scaling operations on image data. We can only duplicate bits as powers of two that may not give the true result always. Thus, in attempts to build a true WYSIWYG ( What You See Is What You Get) editor one must also consider such factors and find appropriate solutions.

There are various software available for public use in preparing documents and graphics designs. Most of these software produce a source code that is simple to interpret and compact with respect to the bitmap representations of the same document. This code is later interpreted (usually by printer drivers) and corresponding listing is obtained from the printer. This approach is useful for being device independent and also for ease of changing and debugging documents. There are two widely used language types for this purpose, Postscript and Prescript. The programs written in these language types can be directly sent to printers that know how to interpret these languages and produce identical results (as far as quality is not concerned) on different printers. Postscript is a stack oriented language (like FORTH) and uses postfix notation. For example, *2 2 add* means  $2+2$ . You can create your bitmap by using three different painting operations defined in postscript: one-dimensional paths, two-dimensional sampled images, and text. It is also necessary to mention that postscript supports intermediate gray levels by digital halftoning meaning that the image is divided into cells where each cell consists of many pixels and can be filled with different patterns to produce a gray level illusion effect. Prescript is based on the same ideas but uses prefix notation instead. The problem with these languages is that they require a software interface to convert users' document description into a program in one of these languages. They are very difficult to use and debug. Another point is that they are simply used for defining a page of listing. Consequently, when you change your document you will have to recreate the whole program again. Moreover, since these languages define documents page by page we cannot expect them to help us in representing relations which may get across

page boundaries quite easily.

## 2.2 DOCUMENT ARCHITECTURE MODEL

In the ODA standard, a *document* is a structured amount of text that can be interchanged either in image form for display or in processible form to allow later editing. Text consists of graphic, geometric and photographic elements and some additional control information.

### 2.2.1 Document structures and objects

In ODA, every document has a *logical structure* defining the hierarchy of *logical objects* such as titles, paragraphs, figures, etc., and a *layout structure* giving the hierarchy of *layout objects* like pages and columns. In our system this hierarchy is defined by the users by setting up relations between data items. Hierarchical relations are rigid and not all relations can be shown by using a hierarchical relation set. In our approach, however, every relation is independent of the other relations that can be defined among the same items, unless they are purposely defined to be dependent on common attributes.

The logical order of a document in ODA is primarily hierarchical and sequential. A hierarchical order is set up, for example, by paragraphs, illustrations, and footnotes as constituents of subsections. A sequential order exists among the objects of the same hierarchical level. For example, a contents list is followed by the document body. This logical structure is a *tree structure* with the logical objects forming the nodes in this graph.

Before a document can be imaged, its layout structure must be established. The graphic elements of the content associated with the logical objects must be arranged within certain layout areas that represent the layout

objects. During imaging, these areas are then mapped onto a physical presentation medium. The layout structure is also a tree structure.

Objects have *properties* and *relationships* which are expressed by *attributes* assigned to them.

Besides the *hierarchical relationships*, which link objects to build up a tree structure and are expressed by the attribute *references to subordinate objects*, there may exist *logical logical relationships* among logical objects that extend across the logical structure, and *layout-layout relationships* among layout objects. Typical logical-logical relationships are, for instance, cross-references to figures or footnotes. A layout- layout relationship is the overlay order of intersecting blocks.

There are also *logical – layout* relationships to control the layout process given as attributes of logical objects.

### 2.2.2 Content portions

The architectural model distinguishes between composite objects and basic objects. *Composite objects* consist of components that may be other composite objects and/or basic objects. *Basic objects* are at the lowest hierarchical levels, and it is only through them that *content portions* are directly associated by means of the attribute *references to content portions*.

### 2.2.3 Object types and their characteristics

Each object in the interchanged data stream is represented at least by the attributes *object identifier* and *object type*. The *object type* assigns additional attributes that may be applied to objects of that type.

ODA distinguishes the logical object types *document*, *composite logical*

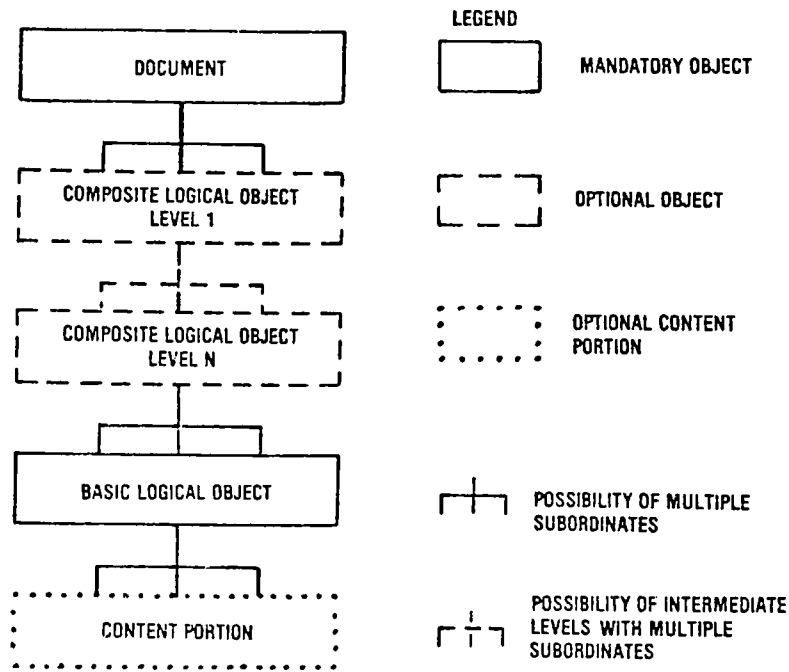


Figure 2.2: Logical object structure.

*object*, and *basic logical object*. Figure 2.2 shows how logical objects form a logical structure and Figure 2.3 shows how a layout structure is formed by layout objects. Note that all of these functions are batch processes and must be performed each time an update is performed. In our system, however, only the changed attributes are reevaluated and the users can see the changes almost instantly and this property makes it superior to a batch processing system. The set of attributes are also predefined in ODA and users cannot insert their own attributes into the document and thus, are not able to define their own relations. Definition of all of these relations are dynamic in our system, and users can define attributes and relations by using either a menu driven system or a very simple language where none of these facilities exist in ODA. All of these deficiencies of ODA are due to the fact that it is defined on a sequential, non-interactive process and a strict hierarchy must exist in the document structure.

The layout object types are *document*, *page set*, *composite page*, or *basic*

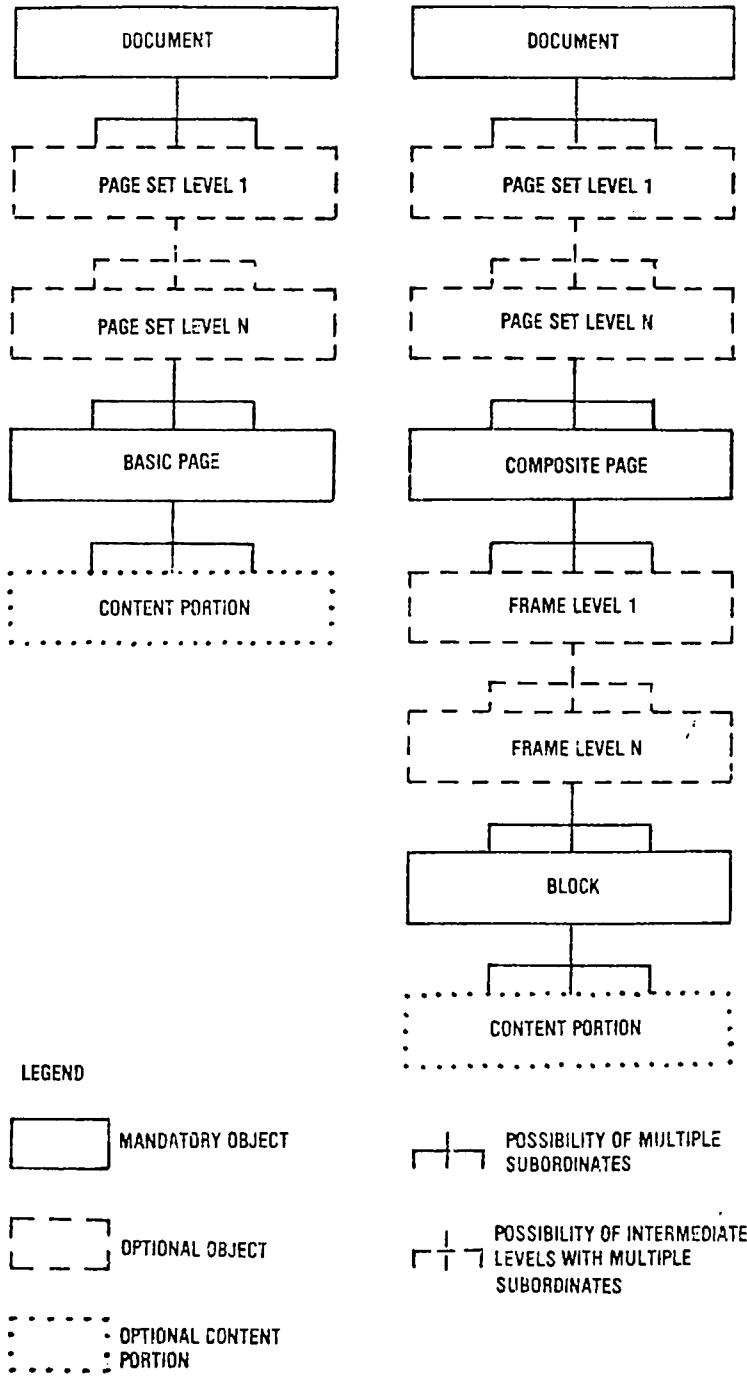


Figure 2.3: Layout structure.



*page, frame, and block.*

An object of the *page set* type is a composite layout object that comprises one or more pages and/or one or more subordinate page sets, which need to be identified as a group. An example is a page set with a title page and continuation pages, either one having different layout areas.

A *page* is a rectangular area that corresponds to a unit of the presentation medium. It is the reference area used for positioning and imaging the content of the document. The size of a page may be smaller than, equal to, or greater than the size of the corresponding unit of the presentation medium. If a document's content is of a single content architecture, the basic layout objects can be of the *basic page* type. In the case of several content architectures, basic layout objects must be of the block type, and pages are then of the composite page type.

A *frame* is a rectangular layout area within a page or within a frame of a higher hierarchical level. Frames define boundaries within a page for the layout of the content; e.g., they can represent header, column, and footer areas.

A *block* is a rectangular area containing one or more content portions of the same content architecture.

Each page, frame, and block has an orthogonal *coordinate system*. Its origin is the object's top left-hand corner. The vertical axis coincides with the object's left edge and the horizontal axis with its top edge. All frames and blocks are positioned relative to the coordinate system of their immediately superior object and are entirely contained within the area of that object.

Within a page, frames and blocks may be positioned in such a way that they *intersect* partially or fully. In this case, their overlay order is given

by a sequence specified in the attribute *imaging order* of their lowest common superior object. Intersecting layout objects have a property attribute *background texture*.

## 2.2.4 Object classes and object definitions

Objects of the same type with additional common characteristics can be grouped into *object classes*. Logical object classes with objects of the *basic logical object* type are, e.g., the classes *paragraph*, *footnote*, and *figure title*. Examples of the layout object classes of the frame type are the classes *header frame*, *column frame*, and *footer frame*.

An object definition is notified by *definition identifier* attribute and its class is specified by a *reference to object definition*. These attributes, by means of constant or variable expressions, can specify the below four types of rules.

1. Property rules
2. Relation rules
3. Content rules
4. Construction rules

These expressions, unlike our system, can only define relationships among text items. Defining relations among graphic and image items is not defined. Even though some positional relationships can be given by arranging layout objects this cannot be done for more complex relations like, color selection, image rotation, colormap selection, etc.

## 2.2.5 Document classes and document definitions

Similar documents can be grouped into a *document class*. Document classes are classes such as *business letter*, *report*, *memo*, and *order form*. Like other classes, document classes are not standardized. They can be defined by the application by means of *document definitions*. A document definition has to contain the definitions for all objects that are allowed to occur in a document of that class.

## 2.2.6 Overall document architecture model and document profile

A document may contain a logical structure and/or layout structure with text content. Description of its document class can be given, as well. A document profile also exists for saving information necessary for manipulating the document, such as the structures and definitions in the document and some descriptive information for editing, indexing, filing, etc. Figure 2.4 shows the overall document architecture model as defined in ODA.

To the best of our knowledge, ODA is the closest system to our application. It is, however, rigid and relation definitions are only applicable to textual data. In the following chapters, our system is going to be defined mostly with respect to the techniques used in breaking this rigidity and providing relations between various data types.

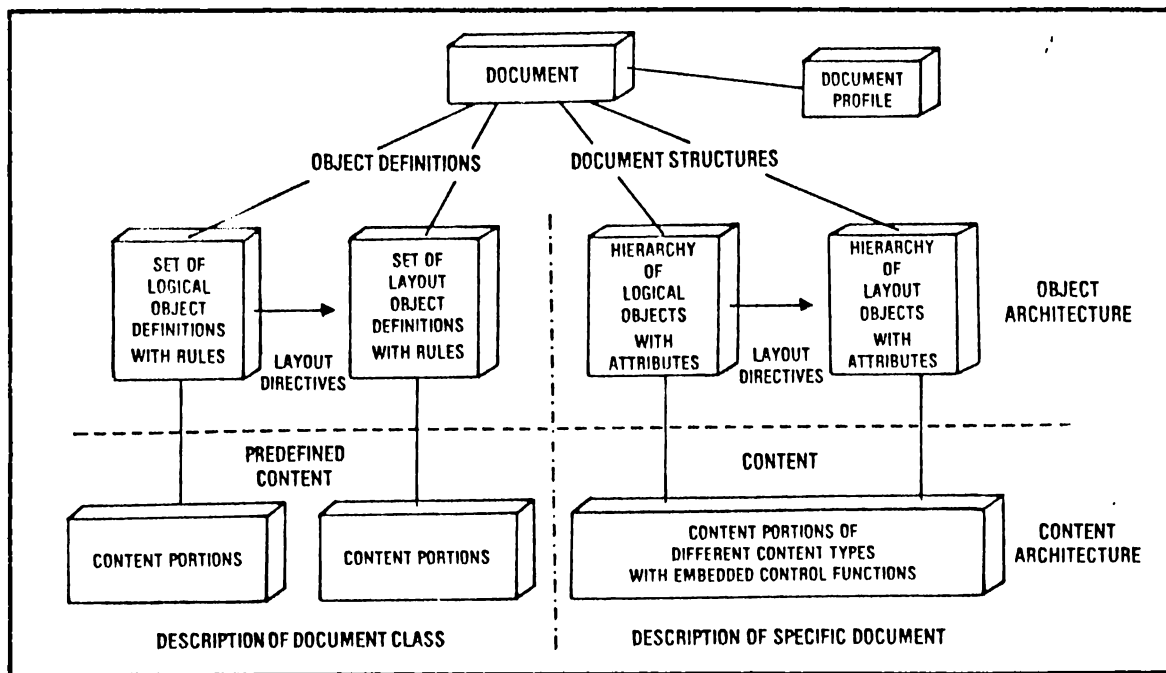


Figure 2.4: ODA document architecture.

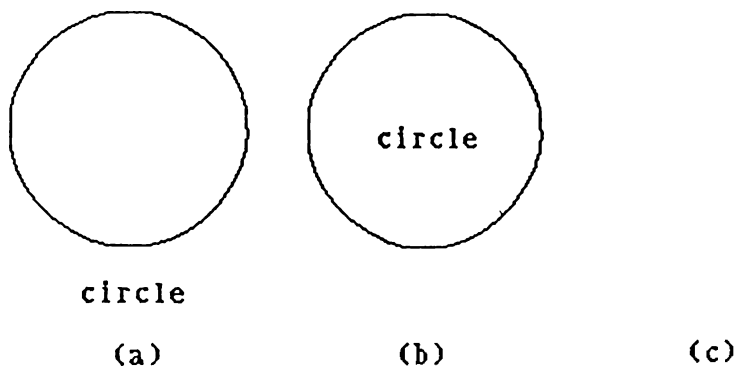
## 3. RELATIONS BETWEEN DATA TYPES AND DATA MANIPULATION

### 3.1 RELATIONS BETWEEN DATA TYPES

When various data types are brought together in an editor, it becomes necessary to define procedures for relating the attributes of one type to another.

An example of such a relation is the visibility of data items. It might be desirable to relate the visibility of text data *circle* to the visibility or existence of the graphical item circle so that the text *circle* appears only when the circle is drawn (figure 3.1). Another possibility is to define the position of the text *circle* to be inside the circle. Centering the text data in the circle and expanding or shrinking the circle as the text inside it expands or shrinks are also possible relations to be defined (figures 3.2 and 3.3). The availability of such relations require a highly dynamic and efficiently manipulated data structure for both the data and the relations.

Definition of these relations, and possibly many more, further requires a rather complex language to be employed. This language can be used by the user to define the relations explicitly or a user interface for automatically selecting some of these relations can be provided. Of course it is not possible to define all relations via the user interface. In order to simplify the selection operations, a default set is defined for each environment and the users will have to deal with the language or nested selections only when they want to



- ```
(a) Text_X= 'circle1 'Circle_X \ival \textwidth 2 / -
    Text_Y= 'circle1 'Circle_Y \ival 'circle1 'Circle_R
        \ival + \textheight + 5 +
(b) Text_X= 'circle1 'Circle_X \ival \textwidth 2 / -
    Text_Y= 'circle1 'Circle_Y \ival
(c) Pixop = 'text1 'Pixop \ival
```

Figure 3.1: The "circle" and the circle with the attribute definitions of the circle. Circle\_X and Circle\_Y give the center of the circle.

use non-default options. Font of a text item, for example, is assumed to be a default font unless the user specifies one with the *Font* attribute.

## 3.2 DATA MANIPULATION

Data need to be updated either due to an addition or correction. Text editing is well defined by the existing text editors that are widely accepted and used. Some of the commonly used editing functions are:

- insertion and deletion of characters
- insertion and deletion of lines
- block operations like copying, moving and deleting blocks of text

There are other primitives for editing text attributes like font selection, blinking, underlining, coloring, etc. These operations can be done either by

|       |                                                                                                                                       |
|-------|---------------------------------------------------------------------------------------------------------------------------------------|
| TEXT7 | Text_X= 'R7 'Box_X \ival 'R7 'Box_W \ival + \textwidth -<br>Text_Y= 'R7 'Box_Y \ival \textheight +                                    |
| TEXT6 | Text_X= 'R6 'Box_X \ival<br>Text_Y= 'R6 'Box_Y \ival 'R6 'Box_H \ival +                                                               |
| TEXT5 | Text_X= 'R5 'Box_X \ival 'R5 'Box_W \ival 2 / + \textwidth 2 / -<br>Text_Y= 'R5 'Box_Y \ival 'R5 'Box_H \ival 2 / + \textheight 2 / + |
| TEXT4 | Text_X= 'R4 'Box_X \ival<br>Text_Y= 'R4 'Box_Y \ival 'R4 'Box_H \ival 2 / + \textheight 2 / +                                         |
| TEXT3 | Text_X= 'R3 'Box_X \ival 'R3 'Box_W \ival + \textwidth -<br>Text_Y= 'R3 'Box_Y \ival 'R3 'Box_H \ival 2 / + \textheight 2 / +         |
| TEXT2 | Text_X= 'R2 'Box_X \ival 'R2 'Box_W \ival 2 / + \textwidth 2 / -<br>Text_Y= 'R2 'Box_Y \ival 'R2 'Box_H \ival +                       |
| TEXT1 | Text_X= 'R1 'Box_X \ival 'R1 'Box_W \ival 2 / + \textwidth 2 / -<br>Text_Y= 'R1 'Box_Y \ival \textheight +                            |

Figure 3.2: Positioning text data inside a rectangle.

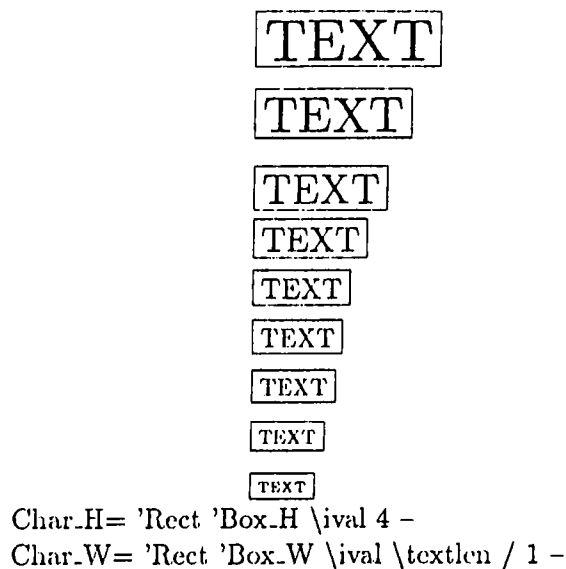


Figure 3.3: Changing font size as covering rectangle shrinks and expands.

selecting options from a menu or by adding control information into the text data. In the latter case we must distinguish between the two modes of display, one for defining these options, second for viewing the appearance of the text under these options.

Graphics data editing is generally more involved because there is the problem of identifying which portion of graphics data is to be edited, the type of editing requested, and the exact positioning and orientation of the item. There are some commonly used techniques for overcoming these problems:

- scale and guideline
- gravity fields
- dragging
- rubber-banding

These techniques are used for editing graphics data as they simplify various operations [15].

Image editing functions include cut and paste operations and painting pixels with arbitrary patterns. Working at a single pixel size is also possible. Scaling of images to expand or shrink and rotating images are also defined operations on image data. It is also possible to convert text and graphics data into image form and manipulate them as an image. In order to facilitate for such editing, insertion (pushing everything on the right by one pixel) and deletion (pulling everything on the right by one pixel) from images in the form of a vertical line (height can be adjusted) is also possible. Cut operations are also enhanced to allow for cutting regions in the shape of any one of the graphical primitives such as rectangles, circles, polygons, and free drawings [4,11].



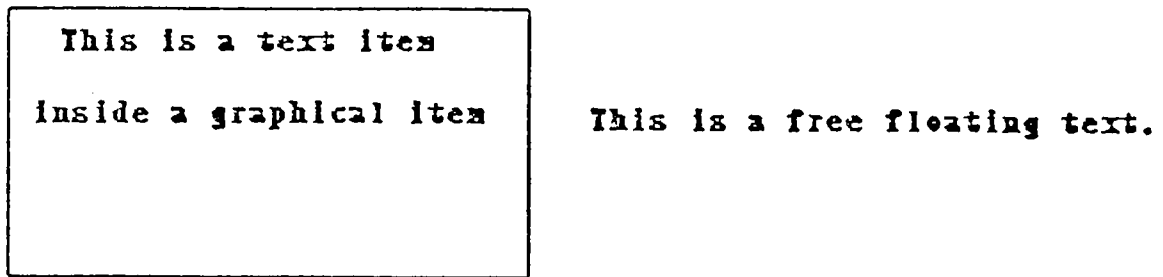


Figure 3.4: Placing text data inside graphical items.

### 3.3 DATA STORAGE

Graphical descriptions are stored as single items each one having its own attribute list. These data items are used mainly for drawing purposes. Another important use of graphics items is for placing textual data inside them. If a text item has the attribute *inside* giving the name of a graphical item, then the position of that text item is calculated with respect to the bounding box of the specified graphics item. If no such attribute is given for a text item, then its position is calculated with respect to the top left corner of the screen or the window.

Storing an image is very much like storing a rectangle since only the coordinates of the rectangular area surrounding the image are saved in the item definition. The corresponding image is stored on a disk file in rasterfile format [16]. When an image is to be edited it is read from the file and stored in a special structure that can readily be transferred to the frame buffer.

Storing and manipulating text data is slightly more tedious since the ordering of text data elements (characters, or groups of characters) must be preserved while inserting and deleting characters. This makes it necessary to keep the item list in correct order and the problem becomes even more complicated when more than one such list must be maintained. Figure 3.4 displays such a situation. The physical way how these data types and their relations are stored is explained in the next section.

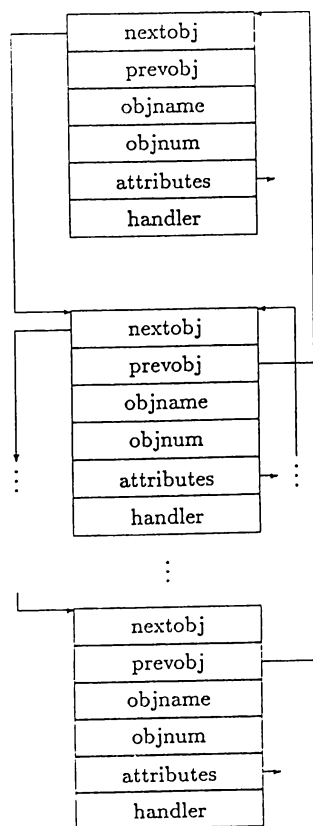


Figure 3.5: The object structure.

### 3.4 DATA STRUCTURES

All of the text, image and graphics items are stored in a doubly linked list of objects, each one representing an item (Figure 3.5). Each object has a unique name and object number which are used for accessing that object. Associated with each object, there is a linked list of attributes giving information about size, location, color, etc. of the corresponding item (Figure 3.6).

Text, image and graphics items are stored as three different object lists for every page of the document. The pages are also stored in a linked list as in Figure 3.7.

In order to explain the use of these structures for defining documents containing text, image and graphics items an example document is shown in Figure 3.8. The data stored in the system for this document are also given and discussed.

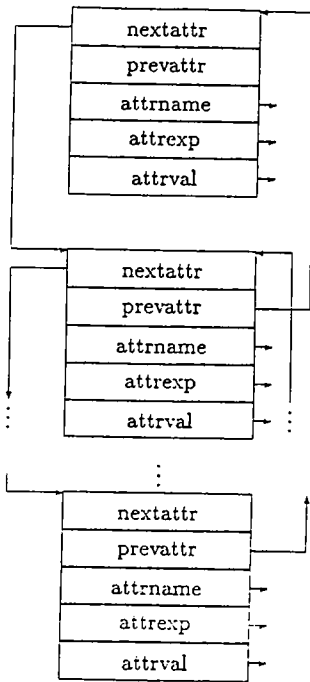


Figure 3.6: The attribute structure.

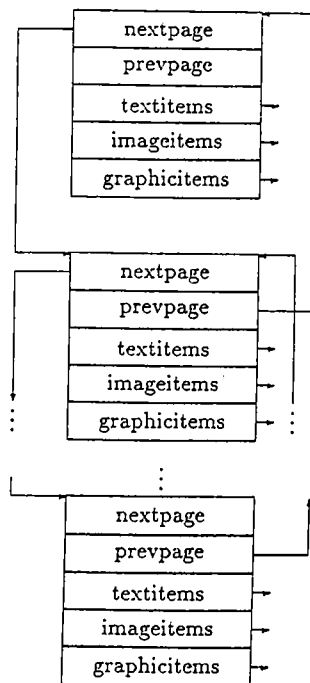


Figure 3.7: The page structure.

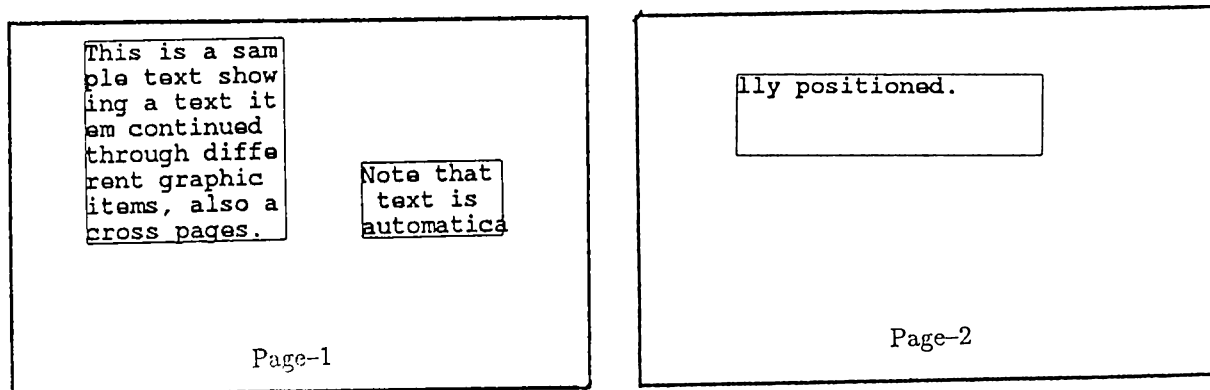


Figure 3.8: A sample two page document.

The sample document consists of two pages. In the first page two rectangular areas are defined for placing text data into them. This is done by assigning the *inside* attribute to the text item. It is also possible to define a text item to be layed out inside more than one graphical area in a predefined order as it is done in the first page. This property is defined by assigning the *intext* attribute to the graphical items which are to include the given text item. This attribute gives both the name of the text item and the order of the graphical item. Please note that, all of these attribute definitions are internal details of the system and the user does not need to know the definition of these attributes that would require the users to find out the internal names of the text items, avoid duplicate numbers etc. Menu selections using the mouse are provided for defining these attributes automatically for any given pair of text and graphics items.

Continuation of a text item across pages is also possible and can be achieved in the way mentioned above with the only difference that the text

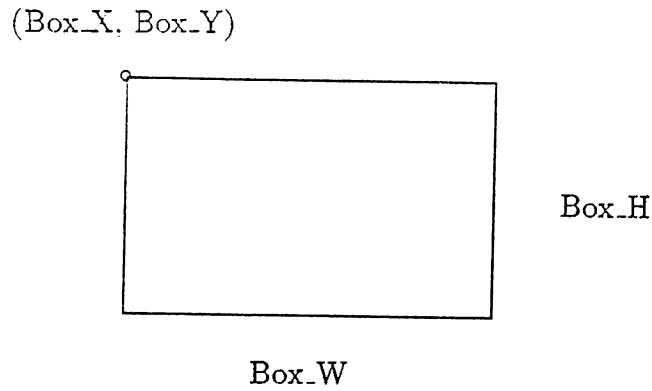


Figure 3.9: Attributes of a rectangle.

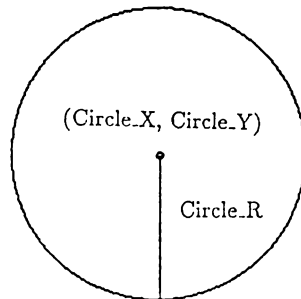


Figure 3.10: Attributes of a circle.

and graphics items reside on different pages. The system, while displaying a text item, checks all of these attributes and performs appropriate adjustments automatically. It is also possible to define text items that are free from all of these attribute definitions and such items can be placed anywhere on the document without having to put them into a graphical item.

Graphics items are simpler to display relative to text items. They simply contain attributes for defining the parameters of a graphical shape. Currently, the system recognizes lines, rectangles, circles, ellipses and polygons. Some of the attributes used for describing these shapes are given in figures 3.9–3.11.

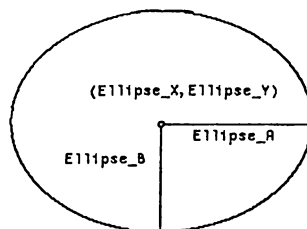


Figure 3.11: Attributes of an ellipse.

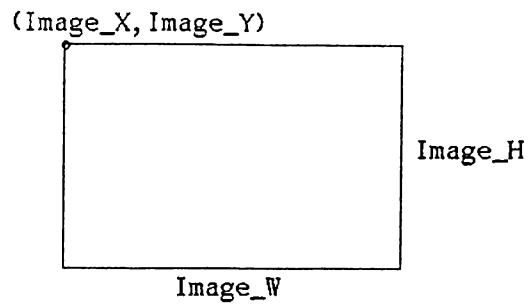


Figure 3.12: Attributes of an image.

Image items are restricted to occupy a rectangular area on the document. The upper left corner, width and height of this area are defined as attributes of an image item. The actual image data is displayed to the user inside this rectangular area and stored in a disk file using the rasterfile format[16] (Figure 3.12).

## 4. SESSION CAPTURE, ARCHIVING AND BACKUP, UNDO, REDO

In order to allow the users to save their work and also as a safeguard against system failures, facilities for storing text, image, and graphics data incrementally are implemented (Figure 4.1). UNDO and REDO are two main operations that are available and give a lot of flexibility to users while performing editing on their documents.

Modern interactive systems have UNDO and REDO commands for recovery (e.g. database management systems). The UNDO command causes the effects of the last edit operation to be removed and the previous state to be restored. The REDO command simply performs the opposite function of UNDO, and does the last UNDO command again.

The UNDO and REDO commands are enough to have a complete system for backuping. Furthermore, both of them must exist in such a system. In order for the UNDO and REDO commands to achieve the work they are expected to do, they must be supplied with a sufficient amount of information about the type of editing and the data that is affected. One simple way is to save the previous contents of data in a backup file each time an update is performed and restore the data from this file when an UNDO operation is requested. This method is not efficient since it usually requires a large amount of secondary storage and long processing time.

```

\graphicsitems
  'rect {
    'Box_Y 20 \vdef
    'Box_X 20 \vdef
    'Box_H 100 \vdef
    'Box_W 100 \vdef
  } \defitem
\imageitems
\textitems
  'text1 {
    'Font 1 \vdef
    'Char_H 10 \vdef
    'Char_W 10 \vdef
    'Text_Y 16 \vdef
    'Text_X 41 \vdef
    'suntext {This is the first text item.} \edef
    'inside {rect} \edef
  } \defitem
  'text2 {
    'Font 0 \vdef
    '\n 0 \vdef
    'suntext {This is the first line break.} \edef
  } \defitem

```

Figure 4.1: A sample file saved on disk.



A better method is to save the type of the modification done and the part of the data that is minimally sufficient for recovery. When such information is stored in a backup file as updates are being made, it becomes possible to perform UNDO and REDO operations without any restriction. Such a facility is especially useful while performing editing operations on different data types since unexpected results can be encountered quite easily. Furthermore, this facility protects the users from failures that may occur and cause loss of data.

## 4.1 UNDO AND REDO IMPLEMENTATION

Data is kept in two types of lists called *item* and *attribute lists*. All of the updates on these structures are performed by four different routines as follows.

- insert an attribute
- delete an attribute
- insert an item
- delete an item

Each time an update is performed using these routines a corresponding UNDO\_REDO record is pushed to the stack that saves these updates. Some of the information kept in this structure is given below:

- *updateno* : an update may cause several updates in the system and these are grouped together in this field.
- *mode* : can be one of text, image, or graphics.
- *operation*: the type of update done with this record.

- objname : name of the updated object.
- attribute data: for saving inserted/deleted attributes.
- oldimage : the structure in which old image is kept
- newimage : the structure in which new image is stored.

For each update performed by the user, only the related information is saved in this structure. Since a stack of these records is kept in the order they are performed, it becomes possible to have an arbitrary level of UNDO/REDO operations (limited by the virtual memory of the computer). For practical reasons, however, a stack size of a small finite value is imposed by the system.

## 5. USER INTERFACE COMPONENTS

Object oriented programming is a new style widely used in many graphics applications. In order to simplify the programming part of this work and also to obtain a user friendly system the tools of this technique were used in implementing the user interface [17]. This approach has a lot that resembles Smalltalk [5] in the principles and standards of interaction, object definition, etc. but it has been implemented in the C language. Figure 5.1 shows the general screen organization of the system.

### 5.1 WINDOWS

The concept of a window in this paper is a particular subdivision of a screen in which a unique task is running. In order to manage the whole screen effectively a window manager is required since otherwise concurrency problems are sure to occur. In such an environment each window can belong to a different task and simulate a terminal controlling that task. When there are multiple windows on the screen the problem is to define the one that is currently active and will receive the inputs provided by the user (keyboard, mouse, etc.). This problem is solved by defining a cursor attached to the mouse and moving it around the screen. The active window is simply defined as the one that the cursor is currently in.

In the case of overlapping windows the window manager avoids confusions

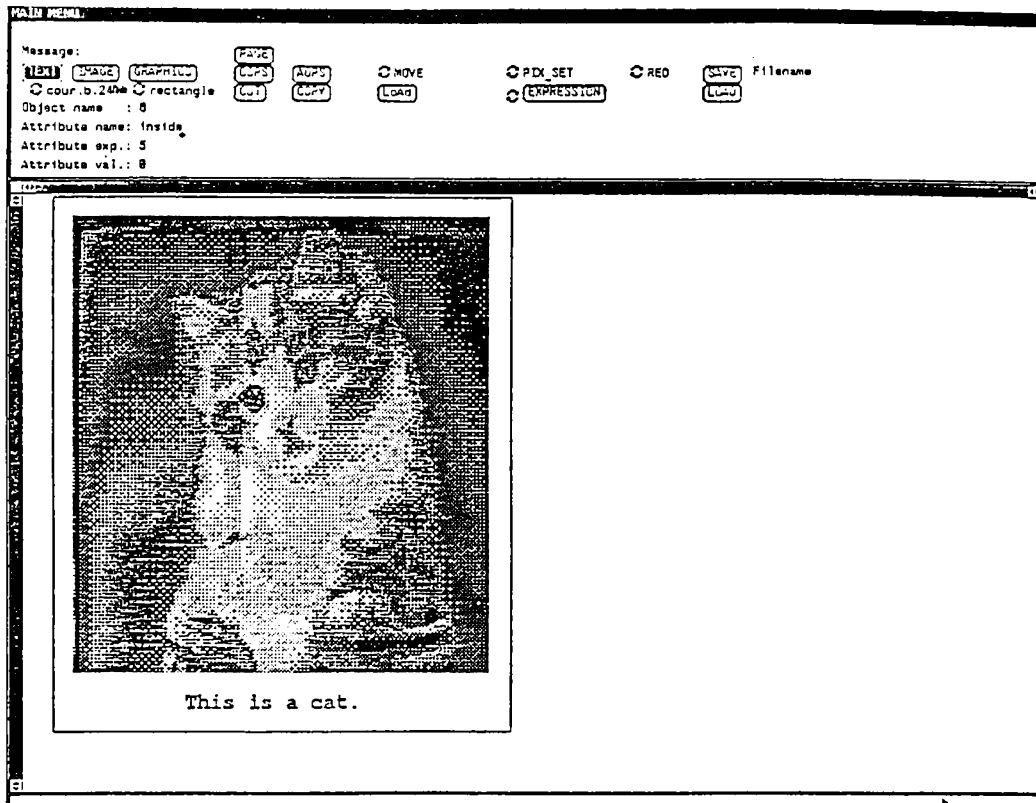


Figure 5.1: A general layout of the screen.

between user interrupts generated through the mouse by selecting the window that is at the top of the hierarchy and has the cursor in it.

## 5.2 MENUS

When a special predefined event is performed such as clicking the button of a mouse at a particular location on a window or on the screen (the whole graphics screen is also a window that is always lowest in the hierarchy), the user might be assigned a temporary object that performs an input through a choice selection mechanism usually referred to as *menu*. Menus contain a list of items in the form of textual, numeric, or pictorial (icon) data. Menus can be used both for selecting options and giving commands to the window manager or individual tasks.

### 5.3 PANELS

Panels include items that can be selected and call special user defined procedures. They can also be used for selecting among a number of items and also for entering text data. Another widely used form of panels is the *slider* item that allows the graphical representation and selection of a value within a range. They are appropriate for situations where it is desired to make fine adjustments over a continuous range of values.

### 5.4 INPUTS, EVENTS, SELECTION

Since there may be a large number of tasks running concurrently in a multiple window environment, and each one may be waiting for an input (or an event in more general sense) for processing, there arises the problem of determining which input (event) is relevant for what window (task) on the screen. Another problem is to notify these tasks of the fact that an event related to them has occurred. The solution to this problem is the Notifier- Selection mechanism that filters out proper events and notifies an appropriate object or module upon the arrival of such an event (Figure 5.2).

This mechanism is also used for managing the screen layout and displaying windows, icons, etc. in the correct form. The functions for changing the hierarchy of windows (hiding, exposing) and for updating the appearance of the screen to conform to the changes made due to previous events are also supplied. In fact, in such an environment each one of the windows, menus, panels, etc. are also predefined objects performing predefined operations when particular events occur. In order to help users to remember the functions that each event performs, help menus and help panels are provided. There are many other events generated by the system and together with user

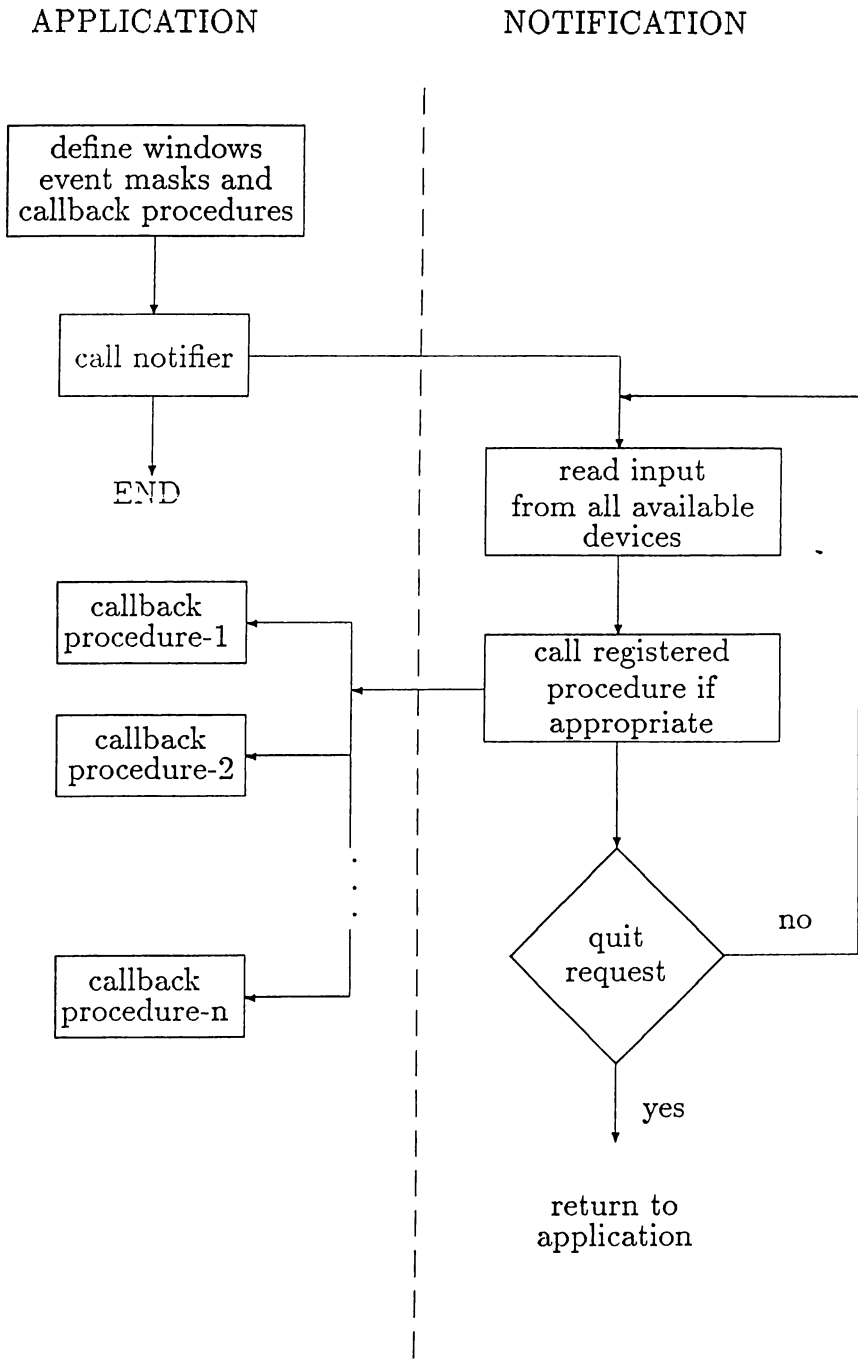


Figure 5.2: Notifier-Selection mechanism.

defined events they make a great number of events most of which are irrelevant to particular tasks. In order to allow the tasks to ignore such events and also to control events that might cause concurrency problems if they are processed arbitrarily, event-locking mechanisms are used.

Some events may cause broadcasts or further events, since many relations are defined between different data types and an update on a data item may propagate its effects onto other related items(Figure 5.3). This makes it necessary to determine what events may occur concurrently with what other events due to the fact that the detection of an event while still processing a previous event may cause the system to process these events incorrectly. The mechanism used for solving this problem is discussed in the next section.

## 5.5 INTERRUPTS AND THEIR MANAGEMENT

The complexity of our system is kept at a minimum by manipulating the events generated by the user within a hierarchy. In other words, once an event has occurred and the system is processing it, further events of the same level are locked and any event that might occur now only is trapped by the current process. If the process, resulting from a single high level and possibly many lower level events, has caused some relations to be altered, or has generated the need for a relation function to execute, it sets appropriate flags signalling so. Until this process is finished, however, other processes are not notified since the user might undo or further modify the effects of the set of events he has generated. Once a done signal is received for the process, then a chain of other processes might start to execute. While still keeping a lock on further events, the main selection function of the system invokes all other processes that it controls. These contain the data type manipulation functions, relation functions, output formatters [2] and various journal or recording utilities. Once the effect of the user request is finished, only then

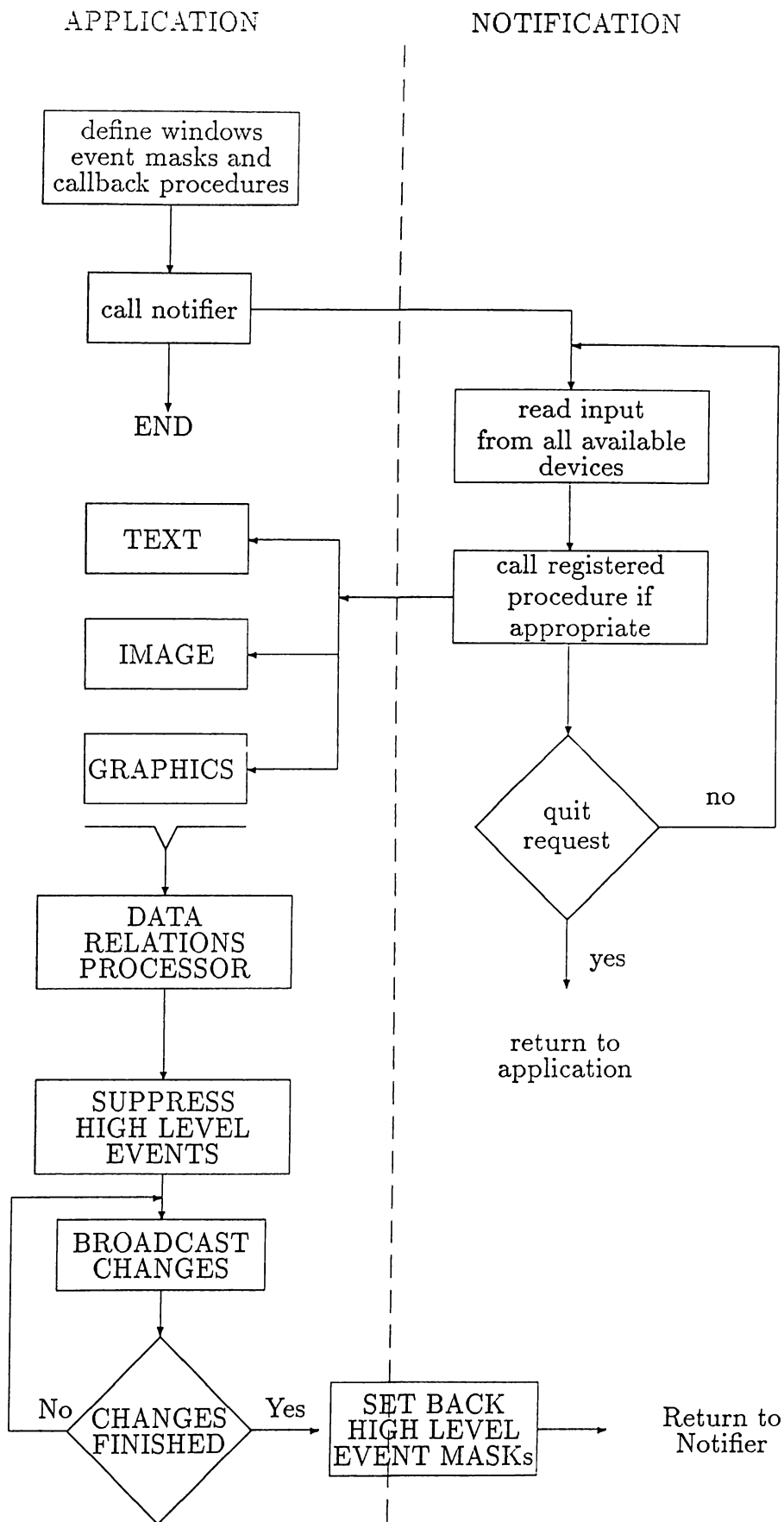


Figure 5.3: Notifier-Selection mechanism with broadcast messages.



the flags are turned off and the lock from the event is removed. This process is called a broadcast and it is during this stage when the relation functions are executed. The menus or a formal language through a text editor only set the definition of these relations in a data template reflecting a particular attribute of an item. Upon the broadcast activities, these data templates are used in their most recent state, imposing the rules of the relation set to the appropriate data type (Figure 5.4). The execution of such a relation function could trigger more flags to be set and if one event changes the state of one data type that is bound to the third one, the whole chain will execute until all of the necessary modifications are made. Much care is taken in order to avoid relation sets that are cyclic thus causing infinite loops that never signals a done flag. The output formatters are used for the reorganization of the display to show the effects of the events.

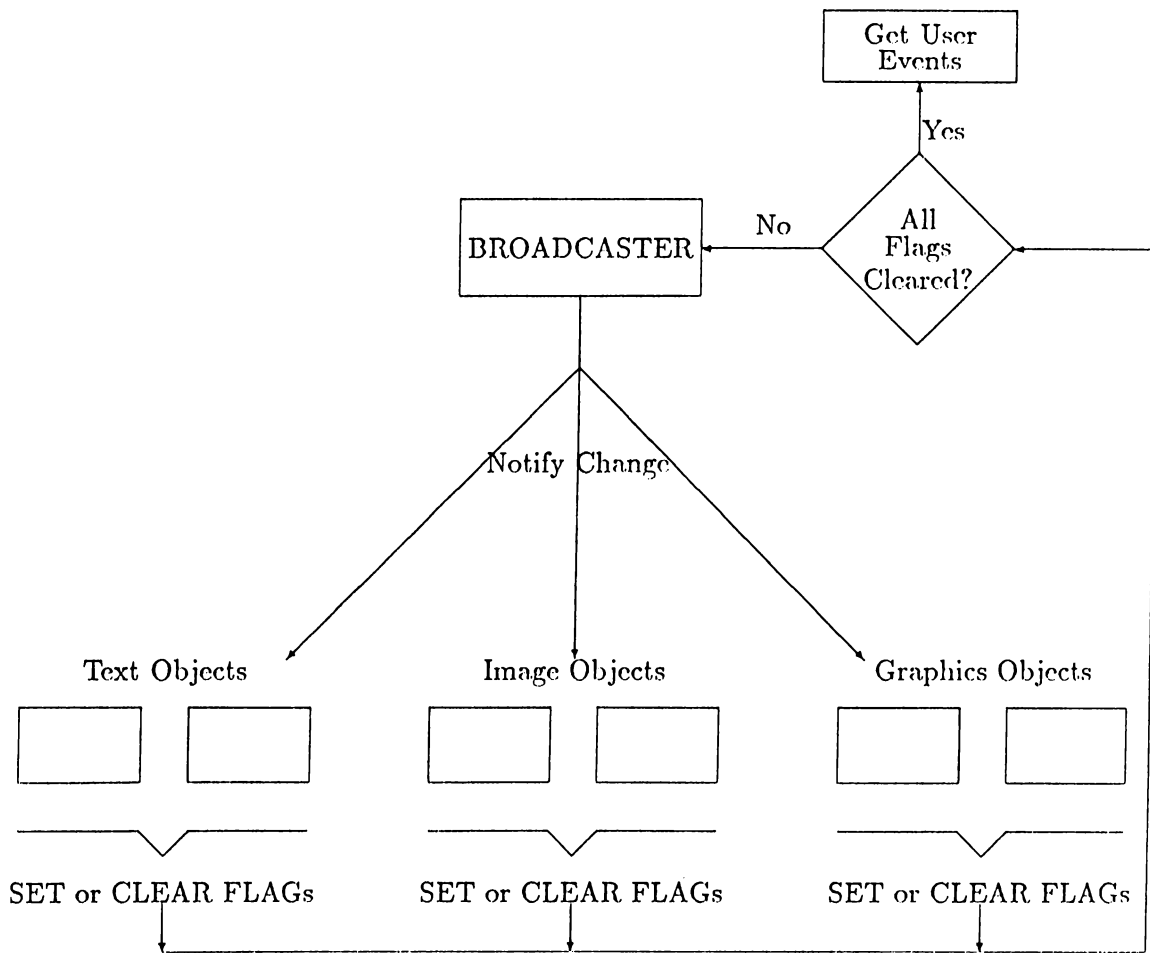


Figure 5.4: Broadcast algorithm for performing necessary updates.

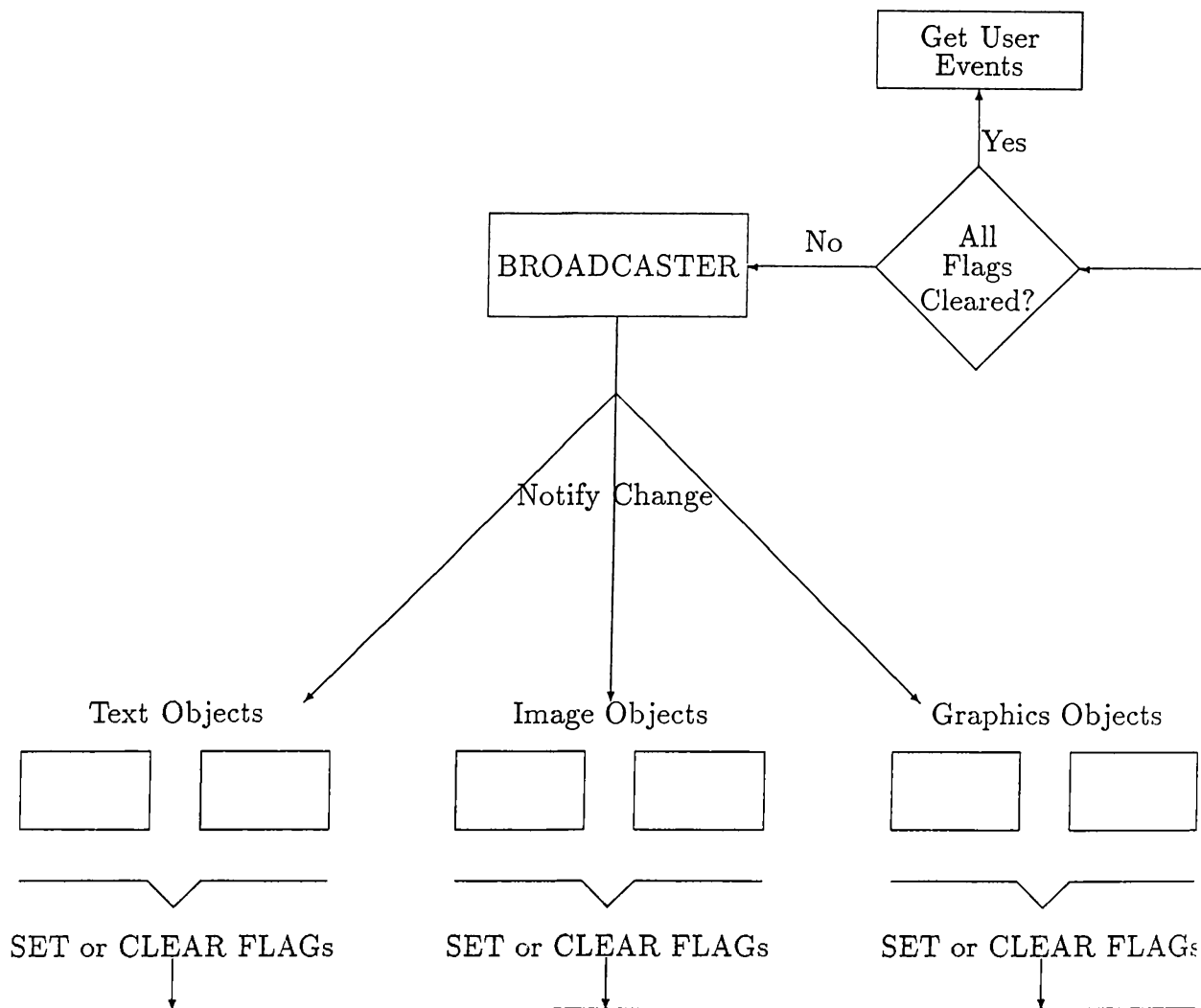


Figure 5.4: Broadcast algorithm for performing necessary updates.

## 6. CONCLUSIONS

The editor designed and implemented in this thesis enabled us to create and edit complicated documents containing text, image and graphics data items. The use of relation definitions between data items has also proved to be very useful and it was observed that a great amount of work could be saved with this technique. These relations can be used for designing layout areas for text, image and graphics windows with respect to each other. References to parts of a document such as figures and tables are also possible and relieves the user of tedious and error prone updates especially in large documents such as articles, theses, books, etc.

The implementation discussed in this thesis can be improved by considering a more enhanced set of text editing functions and some utilities for automatic generation of *table of contents*, *list of figures*, *index*, etc. The speed of the system can also be increased by using shorter attribute names or having a standard set of parameter names valid for any graphical item. Adding zooming and grid size selection facilities for online editing would also be useful. Similar improvements are also possible for graphics and image editing parts of the system. More powerful implementations of the editor for each type of data will increase the productivity of the system, as well. It would also be a good idea to allow the users to enter text and graphics data in image mode and save them as image data. The data defined in this way can be edited only by using image editing functions later. This, however, makes it easier to design simple figures which can be easily redrawn if an

update is needed.

Finally, the implementation of this system and its use has shown the possibility and usefulness of the relations in a document editing system. The graphical user interface and its capability to display the results of an update on the screen as they are being performed is also a very useful property and makes this system superior to batch oriented editors. The possibility of UNDO/REDO operations and journaling and backup facilities are also useful features especially in a multi data type document with possibly very complex relations.

## A. OPERATIONS THAT ARE CURRENTLY AVAILABLE IN THE SYSTEM

This system is designed for efficient and easy definition and manipulation of text, image and graphics data that can exist in a document at any place and in any combination. The main difference of this particular system from other similar systems is its capability of defining relationships among any given set of data items. These relationships are implemented as attributes defined by giving expressions involving attributes of other items.

Another feature of this system is its graphical user interface enabling rapid update and definition operations on the data items. This feature allows visual interaction techniques to be utilized for showing the results of these operations instantly on the screen.

In order to simplify the number of steps required for making an update or definition, the system is assumed to be working on a single data type at any time. The selected data type (i.e., text or image or graphics) is displayed to the user on the screen and can be changed upon user request.

The type of operation to be performed by the user must also be selected, as in the data type selection, to be one of the below operations:

- move

- resize
- edit
- define
- select

All of these operations are self explanatory and their functions are discussed in the following paragraphs while explaining the operation logic of the system on each data type.

## A.1 TEXT OPERATIONS

### A.1.1 Moving text items

This operation updates the *Text\_X* and *Text\_Y* attributes of a text item so that the text data will be placed on the screen to the position where the mouse currently points.

### A.1.2 Resizing text items

This is an important facility of this system that does not exist in most of the other similar systems. With this operation users can change the height and the width of a character (or a group of characters) to meet their requirements. To enable such a function, special fonts are previously defined in terms of polygons and Bezier Curves. Then, it becomes possible to perform width and height changes by simple graphical scaling functions. Rotation and shearing of characters are also possible, though not implemented.

### **A.1.3 Editing text data**

In order to allow users to update previously entered text data or to correct typing errors, a simple set of text editing functions exist in the system. These include:

- inserting characters
- deleting characters
- cutting a block of text
- pasting a block of text
- changing font of a block of text
- etc.

### **A.1.4 Defining text items**

This operation allows text items to be defined either for placing inside a graphics item or as a stand alone data item.

### **A.1.5 Selecting text items**

Since there may be more than one text item in a document it becomes necessary to select one of them as the subject of further editing operations. In other words, this operation simply changes the current text item as to be the one pointed by the mouse.



## **A.2 IMAGE OPERATIONS**

### **A.2.1 Moving image items**

This operation updates the *Image\_X* and *Image\_Y* attributes of an image item to move the top left corner of its image window to the position of the mouse.

### **A.2.2 Resizing image items**

This operation changes the *Image\_W* and *Image\_H* attributes of an image item so that the width and height of its image window are adjusted appropriately.

### **A.2.3 Editing image items**

The system provides many standard painting patterns to the users so that one of them can be selected and copied on an image window applying a bit level logical operation such as, OR, AND, XOR, etc. Selection of a color is also possible for a colored image to be painted with a special color such as, red, green, blue, etc.

### **A.2.4 Defining image items**

This operation allows the users to determine a rectangular area that will include the image data. It also creates appropriate data structures for storing the image data in memory and in a disk file.

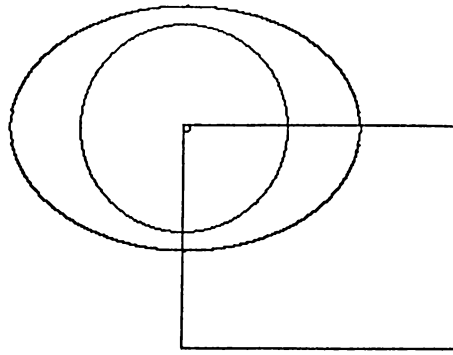


Figure A.1: The result of moving graphics items to the same point.

### **A.2.5 Selecting image items**

This operation selects one of the image items as the current image item so that further image editing operations are clipped against the boundaries of its image window.

## **A.3 GRAPHICS OPERATIONS**

### **A.3.1 Moving graphics items**

This operation is more complicated because the attributes to be updated depend on the type of the current graphics item. Figure A.1 is given to demonstrate this fact.

### **A.3.2 Resizing graphics items**

This operation again depends on the type of the current graphics item. The below figure explains the result of a particular resizing operation on each graphical shape.

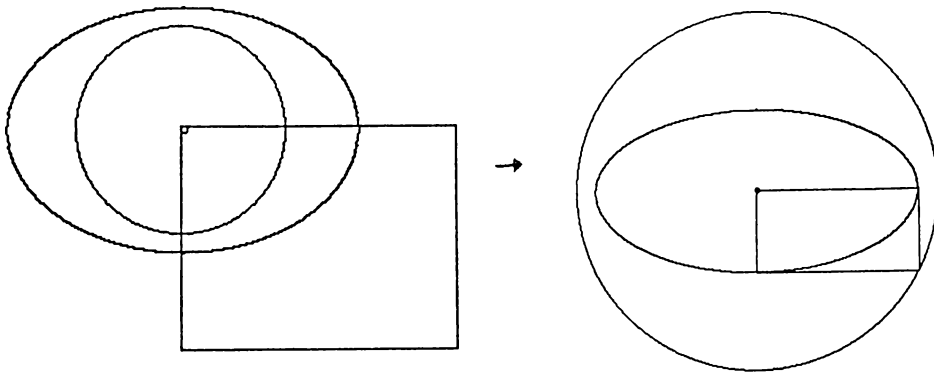


Figure A.2: The effects of resizing graphics items.

### A.3.3 Editing graphics items

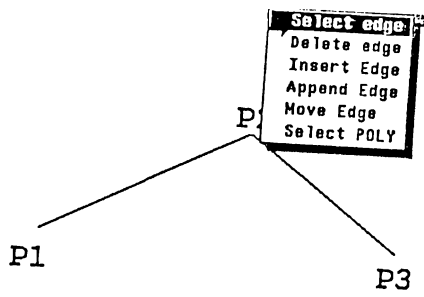
This operation is used only for editing polygons since other graphical shapes can be put into desired shape using *move* and *resize* operations.

### A.3.4 Defining graphics items

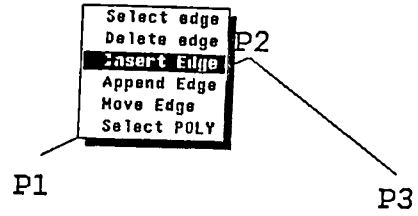
This operation requires the users to select the type of the graphical shape to be drawn. Then, these graphical shapes are drawn on the screen using the dragging technique until the user completes the definition.

### A.3.5 Selecting graphics items

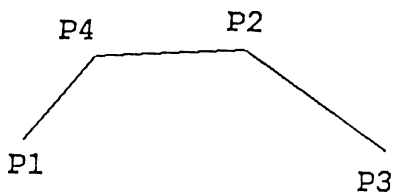
This operation can be used for selecting any one of the graphical items to make it the current graphics item.



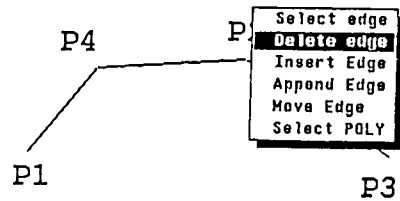
Select edge P2.



Insert edge before P2.



After edge is inserted.



Delete edge P2.



The final polygon.

Figure A.3: Editing operations on polygons.

## B. The Language of The System

This system uses a postfix format language for defining the expressions giving relations between the items of a document. Each expression is evaluated by the system using a built-in interpreter. The commands in an expression are evaluated from left to right and the results are pushed to a stack. With this mechanism other commands can fetch their parameters from this stack. Looping and conditional statements are also available that allow for very complex relations to be written. The postfix format of the language makes the interpreter very simple and new commands can be added very easily. Some of the mostly used commands are explained below:

- `\ival: '<item name> '<attribute name> \ival`

This command finds the value of the item name of which is given and pushes the result into the stack.

- `+, -, *, /` : perform usual arithmetic operations on integer data popped from the stack.

- `\if { <if not negative> } {<if negative> }`

Pops a value from the stack and performs one of the next two expressions depending on the value of it.

- `\loop { <any expression> }`

Evaluates the next expression while the result of it is not zero.

The language has some simple conventions such as string parameters are preceded by single quotation marks and the commands start with the backslash character. There are some other commands used by the system, but they are not needed for writing usual expressions. These functions can be learned from the source listing of the interpreter very easily in case that one needs to use them.

## REFERENCES

- [1] *Computer Graphics Virtual Device Interface*, document X3H3/85- 47, X3 Secreteriat, CBEMA, 311 First st. NW, Suite 500, Washington, DC 20001, 1985.
- [2] Coşar, A., Özgüç, B., *Text, Image, Graphics Editor*, SERC Report:CIS-8803, 1988.
- [3] Coutaz, J., *Abstractions for User Interface Design*, Computer, Vol. 18, No. 9, Sep. 1985, pp. 21-34.
- [4] David Shuey, David Bailey and Thomas P. Morrissey, *PHIGS: A Standard, Dynamic, Interactive Graphics Interface*, IEEE CG&A, August 1986.
- [5] Dawson, B. M., *Introduction to Image Processing Algorithms* , BYTE, March 1987 pp.169-186.
- [6] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
- [7] F.R.A. Hopgood, D.A. Dure, J.R. Gallop and D.C. Sutcliffe: **Introduction to the Graphical Kernel System (GKS)**, 2nd edition, Academic Press (1986).
- [8] Horak, W., *Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization*, IEEE Computer, Oct. 1985.

- [9] *Initial Graphics Exchange Specification (IGES), Version 3.0*, NBS, Gaithersburg, Maryland, Apr. 1986.
- [10] ISO 7942 Information Processing Systems -Computer Graphics- *Graphical Kernel System (GKS) functional description*, ISO Geneva (1985).
- [11] Joseph F. Ossanna, **Nroff/Troff User's Manual**, Bell Laboratories, Murray Hill, New Jersey.
- [12] Kaya, A. and Özgüç, B., *Continuous Processing of Images Through User Sketched Functional Blocks*, **Computer Graphics Forum**, 1988 (in print).
- [13] Keith Y. Cheng, *Microcomputer Graphics and Applications with NAPLPS Videotex*, **IEEE CG&A**, June 1985.
- [14] Knuth, D. E., **The TEXbook**, ninth printing, October 1986, Copyright 1984, 1986 by the American Mathematical Society.
- [15] Lofton Henderson, Margaret Joutney and Chris Osland, *The Computer Graphics Metafile*, **IEEE CG&A**, Aug. 1986.
- [16] Newman, M. W., Sproull, F. R., *Principles of Interactive Computer Graphics*, Ch. 17, Mc Graw Hill, 1979
- [17] SUN Pixrect Reference Manual, Sun<sup>TM</sup> Microsystems, Feb. 1987.
- [18] Özgüç, B., *Thoughts on User Interface Design for Multi Window Environments*, Proceedings of the Second International Symposium on Computer and Information Sciences, Istanbul 1987, pp. 477-488.
- [19] Reed, Theodore N., *Standardization of the Virtual Device Interface and the Virtual Device Metafile*, **Computers and Graphics**, Vol.9, No.1, 1985, pp.33-38.
- [20] **SunCore<sup>TM</sup> Reference Manual**, Sun<sup>TM</sup> Microsystems.



[21] Sun<sup>TM</sup> Microsystems, 1986

[22] W. N. Joy, M. Horton, **An Introduction to Display Editing with Vi**, University of California, Berkeley.