A GRAPHICS SUBROUTINE LIBRARY - MACPAC

THE DESIGN OF MACPAC
- A GRAPHICS SUBROUTINE LIBRARY
BASED ON A DESIGN PHILOSOPHY
FOR THE NEXT GENERATION
OF GRAPHICS PACKAGES

By

HELEN JEAN VRENJAK, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

October 1984

MASTER OF SCIENCE (1984)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: The Design of MacPac - A Graphics Subroutine Library Based on a Design Philosophy for the Next Generation of Graphics Packages

AUTHOR: Helen Jean Vrenjak, B.Sc. (McMaster University)

SUPERVISOR: Dr. P.J. Ryan

NUMBER OF PAGES: vii, 171

ABSTRACT

This paper presents the design of a graphics subroutine library, MacPac, as a contribution to the development of a standard for future graphics packages. The need for a new graphics standard, and hence the motivation for the development of MacPac, is illustrated through a detailed discussion of existing graphics standards and systems. MacPac is based on a design philosophy developed by Mark Green for the next generation of graphics packages. It addresses the hardware and software ideas of the 80's, incorporating and building upon the valuable and tested ideas of a number of existing graphics systems. The design languages used in the development of MacPac were created by Mark Green for the design of user interfaces. This work examines the effectiveness of these languages in the design of a graphics system.

ACKNOWLEDGEMENTS

I would like to express my gratitude to both Professor Mark Green and Dr. Pat Ryan. Professor Green provided the impetus behind this project, along with the design philosophy and design languages used in the development of MacPac. Dr. Ryan was instrumental in the completion of this work. I am very grateful for his patience and guidance throughout the past few months.

I would like to thank all those friends who supported, encouraged, and maintained me throughout this work. The fact that many of these people still speak to me is testimony to their monumental understanding.

Finally, I would like to thank my family for their continuous support, encouragement, and faith in me. Special thanks go to my father for his patience, endurance, and (very importantly) skill in editing this paper. Last, but far from least, my husband, Milo. Thank you, Milo, for everything. I dedicate this work to these people.

CONTENTS

Chapter 1	**Introduction**	1
Chapter 2	**Graphics Packages Today**	5
2.1	The Current Standards - GKS and Core	7
2.1.1	Methodology	7
2.1.2	GKS and Core - Description and Comparison	8
2.2	Mainstream Methodology Revisited	14
2.2.1	Portability and Device Independence	14
2.2.2	Separation of Input and Output	16
2.2.3	The Synthetic Camera Analogy	17
2.2.4	Resource Management	18
2.3	Resource Management Systems	19
2.3.1	Transformations	20
2.3.2	Segmentation	22
2.3.3	Attributes	23
2.3.4	Input	23
2.3.5	Object-Oriented Systems	24
Chapter 3	**Introduction to MacPac**	26
3.1	Motivation	26
3.2	Design Philosophy Behind MacPac	28
3.3	The Design Languages	32
3.3.1	The User Modeling Language - UML	34
3.3.2	The Specification Language - GUSL	35
3.3.3	The Base Language	38
Chapter 4	**The User Model**	41
4.1	Objects	41
4.2	Operators	49
Chapter 5	**Display Considerations**	60
5.1	Objects	61
5.2	Operators	67
Chapter 6	**Specification of MacPac**	90
6.1	MacPac Modules	91

6.2 Graphical Primitives 124

Chapter 7 Conclusions 137

Appendix A UML and GUSL 144

1 Language Structure - Constructs 144
1.1 UML Constructs 144
1.2 GUSL Constructs 145
1.3 The Base Language - Special Forms 148
2 Pre-defined Theory Operators 149
3 Grammars 150
3.1 Base Language Grammar 151
3.2 UML Grammar 152
3.3 GUSL Grammar 154

Appendix B Basic Graphical Entities 157

1 Raster_display Specification 157
2 Transform Specification 158
3 Colour_map Specification 161

Appendix C Compendium of Operators 163

1 Display Manipulation Operators 163
2 Image Manipulation Operators 164
3 Figure_Transform Manipulation Operators 166
4 Input_Device Manipulation Operators 168

REFERENCES 169

FIGURES

Figure 2.1	Mainstream Transformations	9
Figure 2.2	Mainstream Input Classes	13
Figure 4.1	An Example of a MacPac Application	45
Figure 6.1	World Specification	96
Figure 6.2	Display Specification	102
Figure 6.3	Image Specification	115
Figure 6.4	Figure_transform Specification	120
Figure 6.5	Input_device Specification	123
Figure 6.6	Input_event Specification	124
Figure 6.7	Line Specification	127
Figure 6.8	Polyline Specification	128
Figure 6.9	Character and Font Specifications	132
Figure 6.10	Text Specification	134

Chapter 1

Introduction

A salient feature of current computer use is the demand for interactive graphic control. From home computers with their video displays to computer-assisted design and manufacturing facilities, the user expects to manipulate and modify images. He also supposes that this capability will be universal; viz, that what can be done in one environment on a particular system can be done anywhere. This creates a new set of problems for the developers of graphics software systems, and for the applications programmers who use these systems.

The design of graphics software systems, or graphics packages, is an important aspect of computer graphics. Without this software, the rate of production of graphics application programs would be very slow and only expert programmers would be competent to write them. It would be impossible to exploit the potential uses of computer graphics [Newman and Sproull 1979]. However, the constant improvement of both input and output hardware creates an environment of exasperating fluidity for the designers of these interactive graphics packages.

In the early years of computer graphics, a display was typically driven by a device-dependent subroutine package provided by its manufacturer. As graphics terminals became more widely available,

device-independent packages capable of driving a wide variety of display devices came into general use. The main goal for device independence was the promotion of program and programmer portability. Difficulties in transporting application programs between sites, and the necessity to retrain graphics programmers to understand the idiosyncracies of each new graphics installation encountered, were major deterents to potential graphics users [Newman and Van Dam 1978]. A device-independent graphics package used in conjunction with a high level programming language had the potential to alleviate these portability problems considerably. Unfortunately, because of the wide variety among these device-independent packages, portability was still limited to those sites having the particular graphics package used to develop the application.

By the early 1970's, the need for standardized computer graphics practices was widely recognized. The prime advantage of graphics standards is improved program and programmer portability. With a standard, graphics software developers are assured that their application programs will run on any computer or terminal, and graphics users are free to choose among vendors for satisfaction of their software requirements. For the graphics hardware manufacturer, a standard gives guidance for future directions in hardware development and increases the size of the market. It is easier to sell a new computer with its latest graphics hardware when it does not demolish the user's graphics software library. A standard may also solve the problem of picture portability by graphics-containing database portability. When a picture

can be represented as database items, the appropriate representation and storage standards permit the picture to be displayed by different computers on different display devices and result in the same visual image [Hindin 1984].

What emerged were two major standards proposals, the Core system and the Graphical Kernel System (henceforward GKS), tailored to the needs of the vector display hardware of the '70's. As these were being established, significant advances occurred in display device technology; e.g. the microprocessor-controlled raster display device. Our approach to software has also changed with the passage of time. This evolution of both hardware and software raises a question as to the suitability of our present graphics standards for current applications.

This work presents the design of a new graphics package as a step in the direction that software systems must take to best accommodate the hardware and software ideas of the '80's. In this context we shall illustrate the influence of the new technology and the new software approach on the design of graphics standards.

Chapter 2 takes a closer look at what is available today in graphics software. Core and GKS, as the current standards, are examined in some detail. Their methodology is presented, followed by a description, and an examination of some methodological problems that emerged during their implementation and use.

A second stream of graphics packages is then described, these frequently being oriented to raster displays. A common feature of this second stream is that they attempt to control the allocation of available graphical resources. Two examples of these systems (CANVAS and GiGo) are presented. A brief description of object-oriented graphics systems is also provided, as these represent a new and different approach to the design of graphics software.

An introduction to our new graphics system, MacPac, is presented in Chapter 3. MacPac, acronym for McMaster Package, is a graphics subroutine library based on a design philosophy developed by Mark Green [Green 1982a]. This philosophy, which addresses the next generation of graphics packages, combines and enhances ideas from the second stream of systems mentioned above. Chapter 3 examines the motivation for the development of MacPac and introduces the underlying design philosophy. A secondary purpose for this work is also introduced. Green [Green 1981] has developed a notation for the design of graphical user interfaces. It is used in the design of MacPac to determine whether its effectiveness extends to the design and specification of a graphics system.

Chapters 4,5, and 6 present the design of MacPac from an initial user's view to a complete specification of the system.

Chapter 2

Graphics Packages Today

Among the first groups to concentrate on a standardized approach to computer graphics software were the ACM SIGGRAPH Graphics Standards Planning Committee (henceforward GSPC), formed in 1974, and the Graphics Subcommittee of IFIP Working Group 5.2 (henceforward WG5.2), formed in 1975. The early efforts of GSPC produced few tangible results and progress was slow. Although possessing a large body of knowledge on which to base the design of a software standard, the GSPC lacked a corresponding methodology. The IFIP WG5.2 Graphics Subcommittee focused attention on this requirement. In 1976 it sponsored the successful Workshop on Graphics Methodology at Seillac, France.

This workshop, by directing attention to several issues in graphics system design, had a profound effect on the later work of the GSPC [Newman and Van Dam 1978]. The result is the widely known specification for a core graphics system called Core, published in 1977 and revised in 1979. Another group stimulated by the "spirit of Seillac" was the German National Standards body, Deutsche Institut fur Normung (DIN). This Group developed several early versions of the Graphical Kernel System (GKS). The International Standards Organiza-

tion (ISO) Working Group on Computer Graphics, containing representatives from the American National Standards Institute (ANSI), DIN, and other national standards bodies, has since modified GKS to its present version and made it a Draft International Standard [Simons 1983].

The development of graphics software can be separated into two major streams [Rosenthal 1983]. The mainstream, exemplified by GKS and Core, concentrates on viewing, segments, output primitives, and virtual input devices. The second stream, called the RasterOp stream by Rosenthal, attempts to provide facilities whereby graphical resources can be managed. The important concepts here are windows, the refresh hierarchy, and pointing for input. Differences in the two streams may at first appear to reflect the hardware addressed, but the window manager graphics system GiGo (Graphics in/Graphics out), driving storage tube terminals, refutes this [Rosenthal 1983]. "Resource management stream" is perhaps a more comprehensive name for this second stream of software development.

This chapter examines the two streams in more detail. Characteristics of mainstream systems are followed by a discussion of problems encountered in mainstream methodology. Finally, resource management stream systems are considered.

2.1 The Current Standards - GKS and Core

A major force in the development of GKS and the Core System was the 1976 IFIP WG5.2 Workshop held at Seillac. This workshop produced a design methodology that underlies both systems. We discuss this aspect before providing details of the systems.

2.1.1 Methodology

The methodology, set out in SIGGRAPH's 1979 specification for the Core System [GSPC 1979], may be summarized as follows [Rosenthal 1981]:

- The primary objective of a standard is program and programmer portability. It should be possible to move a program from one environment to another without requiring structural changes. Portability also encompasses the concept of device independence, whereby application programmers are insulated from the peculiarities of particular devices.
- Input and output functions should be separated. Input device access should be in terms of the type of value the device returns; this is the concept of virtual input devices [Wallace 1976].
- Two coordinate systems should be supported, the world coordinate system in which the picture for display is constructed, and the device coordinate system in which the data to be displayed are represented. Data in world coordinates are converted to device coordinates by invoking a viewing operation.

- A display file, containing a description of the picture in device coordinates, should be used. Individually modifiable display file segments, each independent of the other, permit manipulation of the picture.
- There should be a distinction between functions involved in maintaining the description of a picture, modelling functions, and those supporting generation of a picture, viewing functions.

2.1.2 GKS and Core - Description and Comparison

The description of GKS and the Core System that follows provides an overview of these systems with respect to some important concepts in graphics software. More detailed information on the Core and GKS may be found in the Status Report of The Graphics Standards Planning Committee [GSPC 1979] and the Computer Graphics Special GKS Issue [X3H3 1984].

2.1.2a Transformations

Viewing, as introduced at the Seillac workshop, is based on the synthetic camera analogy. In this analogy, a scene in world coordinate space is viewed on a display screen as if by a camera located at a single point in that space (Figure 2.1a). The camera handles the transformation of the scene from world coordinates to the coordinates of the display medium.

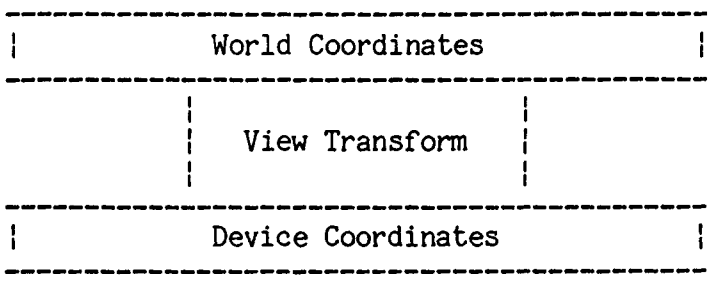


Figure 2.1a - Synthetic Camera Analogy

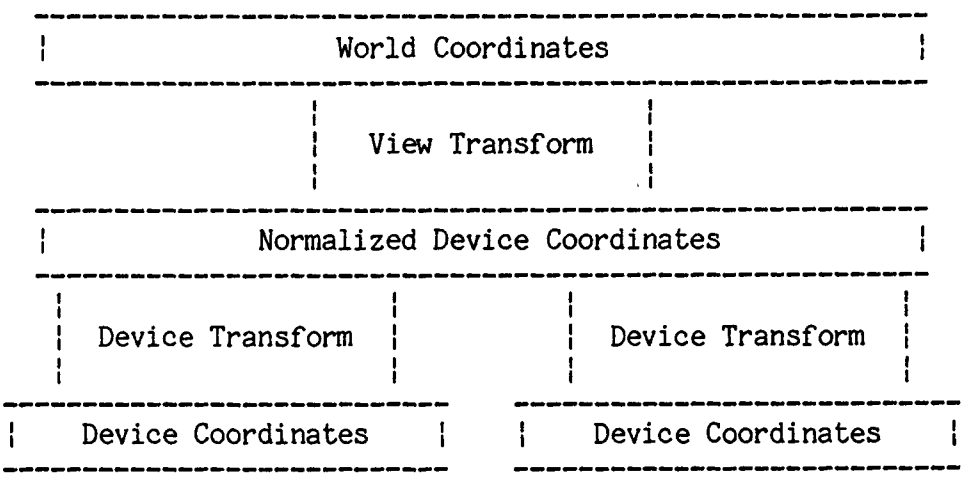


Figure 2.1b - Core System Transformations

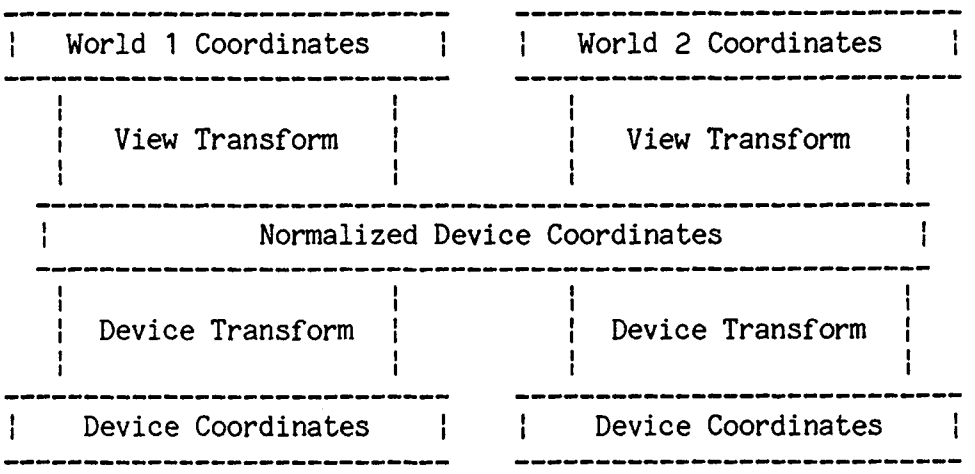


Figure 2.1c - GKS Transformations

Figure 2.1 - Mainstream Transformations [Rosenthal 1983]

In practice, however, this simple analogy cannot accommodate the requirements of either Core or GKS. The stated objective of device independence is extended by both groups to include the ability for the same program to drive different devices simultaneously. To accommodate this, the single transformation from world coordinates to device coordinates required by the synthetic camera analogy is split into a two-stage process. First, world coordinates are transformed to normalized device coordinates. From there a device-specific transformation is used to transform the scene to the coordinates of the chosen device. The Core System supports a single global transformation from world to normalized device coordinates (Figure 2.1b).

In the vast majority of applications, however, several world coordinate systems and/or several different views are required in the generation of a picture. When only a single viewing transformation is available, the application program must recreate the correct viewing transformation, both on output when modifying the display, and on input for the conversion of locator coordinates to the appropriate world coordinates. These considerations led GKS to support multiple normalization transformations (Figure 2.1c). [Rosenthal 1983].

2.1.2b Segmentation

The concept of segmentation handles the problem of selective modification. After a picture of an object has been created, the user may decide to change one area of the object. When the application

makes the desired change to the underlying data structure, an updated picture must be generated. A brute force approach recreates the entire picture, but both Core and GKS avoid this inelegant solution by the use of segments.

The description of an object is partitioned into segments that are individually displayed, and therefore individually modifiable. Each output segment contains a collection of logically related (as determined by the application programmer) output primitives that are to be manipulated as a whole. The application programmer may create segments at will. Initially the segment is in an open state and any primitives output while it remains open become part of the segment. Thereafter, the output primitives contained in a segment are subject only to changes to the segment attributes. In both GKS and Core only one segment may be open at a time. It is worthwhile to note that for these mainstream systems "[segmentation] also addressed the problem of refreshing the picture after changes, storing a description of the picture as output primitives in a segmented display file. The range of permitted [segment] manipulations was modelled on the capabilities of a typical refresh vector display, highlighting, visibility and detectability changes, and segment transformations" [Rosenthal 1983].

2.1.2c Attributes

There are several types of attribute supported by mainstream systems. Global, or output primitive, attributes apply to primitives as they are generated and are not subject to retroactive modification. Attributes of this type may control the geometric aspects of a primitive, such as the height, width, and style of a character, or may simply determine the appearance of a primitive, such as the style (dashed, dotted, solid, etc.) of a line. Segment attributes are assigned to segment by name and may be altered at any time. These attributes, which affect all primitives contained in a segment, control features such as the visibility and highlighting of the segment. [X3H3 1984] [GSPC 1979].

Additionally, GKS supports workstation attributes. An example of this is "SET COLOUR REPRESENTATION", whereby the colour to be associated with a particular colour index on a particular workstation is specified [X3H3 1984]. GKS offers another feature, the bundling of attributes, which is not supported by the Core System. Several non-geometric attributes may be grouped under a single identifier that is an index into a bundle table. The index becomes the single attribute of the primitive under consideration and, as each workstation has its own bundle table, the appearance of this primitive may be different on different workstations [X3H3 1984].

2.1.2d Virtual Input Devices

The input facilities of these mainstream packages are built upon an extended concept of virtual input device. The pure virtual device concept specifies that the only visible aspect of a device is the type of value it returns. In practice it is obvious that virtual devices require other visible aspects to enable control of the operator interface. For example, the ability to control the prompt, echo, and initial value of an input device is required. This extended concept is known as the logical input device concept [Rosenthal 1982].

In the mainstream, each input device is assigned to a class dependent on the type of value the device returns, and is accessed in terms of this class. The six possible classes are listed in Figure 2.2. Additionally, information can be obtained from each device in different modes. The three modes used by Core and GKS are EVENT, SAMPLE, and REQUEST. In EVENT mode the input device creates event records and

CLASSES		
Locator	-->	Position (World Coordinates) + Transformation ID (GKS)
Stroke	-->	Position (World Coordinates) + Transformation ID (GKS)
Valuator	-->	Real Number
Choice	-->	Integer
Pick	-->	Segment Name + Pick ID
String	-->	Characters

Figure 2.2 - Mainstream Input Classes [Rosenthal 1983].

adds them to an input queue. When the application requests input from a device in this mode the first record in the queue is returned to the program. In SAMPLE mode the device is simply polled for its current value when information is desired. If an application requires information from a device in REQUEST mode, it is suspended until this input becomes available. Each device has attributes which specify initial values, restrictions on values returned, and parameters for echo implementation. [Rosenthal 1983].

2.2 Mainstream Methodology Revisited

Throughout their development, the proposals for GKS and the Core System were subjected to intense international review. This review identified certain problems inherent in the methodology. In some areas solutions were found which appear to be in conflict with the methodology, while in others the solutions appear unsatisfactory and remain controversial [Rosenthal 1981].

2.2.1 Portability and Device Independence

Early in the development of these standards proposals, it was recognized that the objective of portability is in conflict with the requirement for high-quality user interfaces. In conjunction with the extended definition of device independence, portability requires that the graphics package be sufficiently general to drive any device. However, to provide for high-quality interaction the application must be tailored, or be able to tailor itself, to the particular hardware

involved. To establish a standard which impedes the construction of high quality user interfaces will tend to reduce the overall quality of applications. Both Core and GKS therefore sacrificed rigid adherence to the methodology to provide mechanisms that allow the application programmer to take advantage of the available hardware in the creation of a user interface.

One partial solution to the problem, adopted by GKS, is to let the application program inquire as to the capabilities of the device it is driving [Encarnacao 1980]. Unfortunately, the application programmer must then anticipate the peculiarities of the all devices to be driven in the future. This involves a large amount of extra code and, because the required path through the code is chosen at run-time, this overhead must be carried whether needed or not [Rosenthal 1981].

Additionally, both Core and GKS provide functions that allow the application programmer access to the non-standard capabilities of intelligent terminals. In the Core System, the ESCAPE function allows the programmer to invoke non-standard function names. In GKS this facility is provided by DRAW, a generalized output function which enables the user to specify higher level graphics primitives. [Arnold 1980].

2.2.2 Separation of Input and Output

Additional methodological problems involve the separation of input and output functions and the concept of virtual input devices. Experience shows that adherence to these concepts impedes the production of high-quality user interfaces. Two important factors in the quality of the human interface are the actions required by the user to create input and the visual feedback received in response to this input. When constructing an application using virtual input devices, however, the programmer simply selects the virtual device corresponding to the type of value desired. Also, when the functions of input and output are clearly separated, the programmer is not provided with the facility to specify what the echo for this input should look like. This places the responsibility for the design of user interfaces on the graphics system implementer, who may not know the application specific meaning of the input required. Another problem is that the application programmer has no idea what the user interface will look like if his program is run at another site, even though the hardware may be identical. [Rosenthal 1981].

The idea of virtual input device has created controversy in yet another area. As Rosenthal points out :

> "The concept of virtual input device tends to segregate input devices into a number of different classes. This is in conflict with accepted wisdom in operating system design, where the objective is generally to unify the various sources of input into a single concept, such as the 'file'. The diversity of virtual devices is particularly confusing because each can be emulated by combinations of the others; there is no real basis for distinguishing them." [Rosenthal 1981].

The seriousness of these problems with respect to user interface quality has led GKS and Core to partially abandon the methodology in this area. Both systems provide facilities which allow the user to exert some control over echoing. Additionally, the Core system has functions for associating input devices into groups and GKS allows for the emulation of one type of virtual input device by another [Rosenthal 1981].

2.2.3 The Synthetic Camera Analogy

The synthetic camera approach to viewing, introduced in the original Seillac methodology, leads to a graphics system that supports only a single transformation. This transformation may be changed during construction of a picture but only the most recent or current transform is known to the graphics system. Unfortunately, the realization of this concept results in problems for both input and output functions. On input, although the user may position a locator device anywhere on the screen, the system is able to transmit meaningful world coordinates only for positions within the last image created, i.e. the image for which the viewing transform is known. To create more than one view of a picture, the transform must be altered and the entire picture reprocessed for each view. As was illustrated in Section 2.1.2a, both Core and GKS encounter difficulties with this aspect of the methodology. GKS partially resolves these problems by supporting multiple viewing transformations. Because only one transformation may be active at any given time, however, this output problem persists.

2.2.4 Resource Management

A major influence in the design of both GKS and the Core System was the idea that a distinction should be maintained between the functions of modelling and viewing. According to the methodology, the graphics system handles viewing. All modelling functions must be handled externally by the application program. Related to this forced exclusion of modelling functions, however, is the mainstream's lack of mechanisms to control the allocation of real graphical resources.

Awareness of the need for graphical resource management was stimulated by the second Seillac workshop in 1979 which stressed the importance of building interactive applications from existing components. To do this successfully requires management of the available graphical resources. It is a way of ensuring that, once an existing component is invoked, it will access only the graphical resources it has been allocated.

Conventional programming languages support a modular program structure that is hierarchical in nature. Parent modules may invoke submodules that may in turn invoke other submodules. To control the management of graphical resources within this structure requires a mechanism whereby the available resources may be allocated in a hierarchical fashion. A graphics package must assign descriptors to all graphical resources. Each part of an application may then manipulate only those resources whose descriptors it has access to. A calling module may pass descriptors for all or part of its graphical resources

onto a submodule. This type of control ensures that no part of the application will have access to the resources belonging to another part. Specifically, no called module will have access to any resources which are not within the control of the module initiating the call. [Rosenthal 1983].

Neither Core nor GKS provides such a mechanism. These systems allow all parts of an application access to the entire view surface. There is no supported hierarchy of pictures on the display and each module can manipulate every picture. Also, any part of the application may access any input device. When input is received, the application must decide to which of its parts that input relates, and handle it accordingly. [Rosenthal 1983].

2.3 Resource Management Systems

We now turn to the resource management stream of graphics systems. Here a consensus of ideas and methodology like that established for mainstream systems does not exist. Many of the graphics systems in this stream were developed for specific purposes and/or in unconventional languages such as Smalltalk [Ingalls 1981], LISP [Sproull 1979], and EDL [Green and Philp 1982]. The resultant variability in these systems makes it difficult to discuss resource management packages as a group. For illustrative purposes, however, two of these systems are presented here: the CANVAS package, developed at Carnegie-Mellon as part of the SPICE project [Ball 1983], and GiGo, an experimental package developed at Edinburgh University [Rosenthal

1981]. Although CANVAS, written in Pascal, was developed for use with raster displays and GiGo, written in C, is used to drive storage tube terminals, the structures of the two systems are remarkably similar. As an aid to comparison, these resource management systems are examined in the same format used to describe the mainstream systems in Section 2.1.2. A brief description of object oriented graphics systems, of which the Smalltalk and EDL graphics systems are representative examples, is included at the end of this section. These systems present an interesting new approach to graphics programming and resource management.

2.3.1 Transformations

There is a significant difference between the two streams in their way of transforming data for display. Mainstream packages use three coordinate systems: world coordinates, normalized device coordinates, and device coordinates. Both GiGo and CANVAS support only two coordinate systems. Pictures described in world coordinates are transformed directly into device coordinates for the desired display.

In GiGo, the Window structure handles this transformation. Specified within each Window structure are two rectangular areas, one in application coordinates and the other in device coordinates. These areas represent a mapping between the two coordinate systems. Each Window also contains information as to which screen the Window is currently active on. [Rosenthal 1983].

When using CANVAS, an application creates a picture by sending output primitives to one or more "canvases". There may be as many canvases as the application desires and each of these is always available for output, even if this output is not to be visible. For a canvas to become visible it must be mapped to a rectangular portion of the display surface called a viewport. This mapping converts canvas (world) coordinates directly to viewport (device) coordinates. A canvas may be mapped to several viewports, but each viewport may display only one canvas. [Ball 1983].

As several viewports may overlap, CANVAS organizes active viewports into a hierarchical refresh tree. At the root of this tree is a special viewport which controls access to the entire display surface. An active viewport may have several child viewports, each describing a patch of the display, but these children will have visible effect only within the portion of the view surface controlled by their parent. A precedence order is established among siblings to handle any conflicts which may arise due to overlap of viewports at the same level of the refresh tree. The only way a viewport may be added to this tree is through a parent. If a module has access to a viewport descriptor, that module may create child viewports within this viewport's display space but may have no effect on any other part of the display. [Ball 1983].

2.3.2 Segmentation

For mainstream systems, segmentation provides both a means of naming groups of output primitives for manipulation and a vehicle for storing a picture description for use in future regenerations. Although GiGo and CANVAS do not support a segment concept per se, they possess other facilities capable of playing the same role.

In the GiGo system, the concept of segment is embodied in the Window structure. Each GiGo Window stores either a list of all graphical primitives contained in the window or a pointer to a routine capable of recreating the picture in the window. In this way, the Window structure corresponds not only to a patch of view surface but to all the graphics in that patch. GiGo Windows can also play the role of segments as symbols. In this capacity the Window contains the graphical primitives of a picture component but is not active on any screen. Other Windows may then reference this Window to incorporate its picture component.

In the CANVAS system the combination of canvas and viewport provides many of the same functions as segments do in the mainstream. Viewports may be added to and removed from the refresh tree just as segments may be made visible and invisible. Raster operations may be performed within a canvas just as segments may be highlighted. Both segments and canvases represent collections of output primitives. Unlike segments, however, canvases are always open. The only picture storage provided by CANVAS is the bitmap; a description of the picture as primi-

tives is not supported. A viewport may store its own complete bitmap and/or the bitmaps of other viewports which it obscures. At higher levels a picture is defined procedurally. When regeneration is required the system arranges for appropriate application code to be invoked. [Rosenthal 1983].

2.3.3 Attributes

Both CANVAS and GiGo support only a single type of attribute. In GiGo, all attributes are Window attributes. They are stored in the Window structure and are applied to all primitives belonging to the Window. Similarly, CANVAS attributes are canvas specific and are applied to all primitives sent to the particular canvas. With this form of attribute control, a caller can create an attribute context for a callee to operate in; i.e. the callee will only be able to affect the attributes of the canvas (or window) whose descriptor was provided by the caller. [Rosenthal 1983].

2.3.4 Input

Neither of these resource management systems support the virtual input device concept used by the mainstream. Instead, all physical devices are mapped into a single input class. For CANVAS this input class, called the Key Event, returns an integer command code, a character, and an integer coordinate pair. The single input class supported by GiGo returns a device coordinate position and an integer code.

The handling of input information is also very different between mainstream and resource management packages. In the mainstream, all parts of an application can access input information from all devices. In contrast, GiGo and CANVAS ensure that the user's input information is accessible only to the routine associated with the window being pointed to. When input is received the GiGo system scans all windows active on the view surface until it finds a window containing the input's device coordinate position. The input routine for this window is then invoked with the input information and the window descriptor as arguments. In the CANVAS system, each canvas has an input queue to which Key Events directed to that canvas are added. A canvas never sees irrelevant inputs.

CANVAS provides a facility whereby events sent to a canvas may be passed up the refresh tree to its parent. This is useful when the application program decides that a certain input cannot be handled at the received level. It is possible to imitate this facility in GiGo simply by calling the parent's input routine directly from the input routine of the child Window. [Rosenthal 1983] [Ball 1983].

2.3.5 Object-Oriented Systems

We conclude our discussion of current graphics packages with a brief overview of object-oriented graphics systems. These systems, which we consider to be part of the resource management stream, represent a radically different approach to graphics programming. In fact,

the differences from the systems we have examined so far are sufficient to preclude discussion of object-oriented systems in the format previously used.

The object is the underlying concept in an object-oriented system. It may be defined as "a package of information and a description of its manipulation" [Robson 1981]. In this way, an object represents both data and the computational processes which manipulate this data. The information in an object is altered by sending a message to that object. The content of the message determines which object processes are invoked and thus the information to be updated. In an object-oriented graphics system, objects are used to structure images on a display. Each object is responsible for a single image and is allocated a portion of the display surface. The object may have an affect on the display screen only within this allocated area. These object-oriented systems support a small number of graphical primitives that are used to directly alter a raster display. [Green 1982a].

This brief description of object-oriented graphics systems is sufficient for the purpose of this work. Although the new ideas seen in these systems must be considered in any future development, the systems themselves are very specialized and are therefore not especially relevant to the development of future standards. If desired, further information may be found in the documentation for the graphics systems of specific object-oriented languages. Two representative examples are the graphics systems for the object-oriented languages Smalltalk [BYTE 1981] and EDL [Green and Philp 1982].

Chapter 3

Introduction to MacPac

Having reviewed some representative examples of the graphics systems available today, we turn to the design of MacPac. The underlying purpose of MacPac is to exemplify an approach that takes advantage of current ideas in hardware and software. From a number of such developments the shape of a new standard graphics package might be expected to emerge.

3.1 Motivation

The progress of technology inevitably requires a corresponding evolution of the control software. Both Core and GKS were developed for the hardware of ten years ago. As an initial stage in the design of the Core system, GSPC surveyed the existing state-of-the-art software. Some prerequisites for inclusion in this survey were that a system be installed at several locations, have an established user base, and be oriented for use with a FORTRAN application program [GSPC 1977]. Given the delay between the time a system is designed, developed, and accepted for use by a number of groups, the systems used as models by GSPC were already years behind the frontier in terms of hardware technology. The survey specifically excluded any system features which "de-

pended extensively on unusual or uncommon hardware features. Thus matrix processors, raster display background/area commands and similar interesting features were excluded" [GSPC 1977].

Thus Core and GKS address the predominant display hardware of their time, the vector display. These devices, having few stand-alone capabilities, are controlled by the host computer and share its memory. In both systems they are driven by a display file. This model is not well suited to the microprocessor-controlled, raster-based displays that are predominant today. Additionally, the current standards do not provide facilities that allow the application programmer to take full advantage of the many capabilities of the modern intelligent terminal.

Further, the approach to producing graphics applications software has changed since the establishment of the current standards. Initially a portable graphics standard was expected to reduce the cost of producing applications software, but this did not occur. In the development of a system the application programmer spends most time on the user interface and application data structures. Tools are becoming available to aid in these tasks; e.g. user interface management systems, graphical databases, special purpose modelling systems, etc. Ideally, any new standard should interface cleanly with these program development tools [Green 1982a].

The current awareness of a need for graphical resource management is another area where our approach to graphics programming has changed. If the graphics package does not support resource management, the application program must assume this function.

Chapter 2 examined several resource management stream systems that address both current hardware and software ideas. In many cases the development of these systems responded to a need that could not be satisfied by the existing standards, or addressed methodological problems encountered in the mainstream. Most of these systems are oriented to very specific systems and/or applications. Many are written in uncommon languages that are not widely used or supported. We need a system capable of fulfilling the needs of a larger audience of graphics application programmers. It should incorporate the valuable features of, and exploit our experience with, existing systems.

3.2 Design Philosophy Behind MacPac

We present MacPac as a contribution to the development of a standard for future graphics packages. MacPac, acronym for McMaster Package, is a two-dimensional graphics subroutine library based on a design philosophy developed by Mark Green for the next generation of graphics packages [Green 1982a]. A summary of this philosophy is provided here.

Green chose the form of a subroutine library as the most viable structure for future graphics packages. This decision was prompted by the lack of success of alternatives such as special graphics languages and programming language extensions. Green also suggests limiting the system to two dimensions. As most applications do not require three dimensions, the inclusion of this capability in the graphics package becomes unnecessary overhead. An add-on subroutine library can usually fulfill the requirements of those applications that demand three-dimensional graphics [Green 1982a].

Two major concepts in Green's design philosophy are the figure and the image. The figure allows the programmer to divide a picture into logical entities that may be manipulated separately. In this sense, it plays the role of the segment in mainstream packages, and of the object, or process, in object-oriented graphics systems. In mainstream systems, however, only one segment is available for output at a time, a segment may not be modified after creation, and segments may not refer to other segments. This makes the mainsteam segment a far more restrictive concept than is desirable. On the other hand, the use of a separate process for every picture segment is far more general than required by most applications. As an organizational entity, the figure falls somewhere between a segment and an object. [Green 1982a].

Each figure is composed of graphical primitives and/or calls to other figures. There is no limit on the number of figures in existence at a given time and no restriction on when a figure may be modified.

Figure modification is achieved either by changing one of its primitives or changing some aspect of a called figure. The primitives of a called figure may be altered, or the transforms that position figures within their parent figure may be changed.

A figure becomes visible as a result of a two-stage process. First, the figure is associated with an image. An image specifies a list of figures and a coordinate space in which these figures may appear. The image is then associated with an area on one or more displays. Several images may appear on a single display. Overlap of images on a display is handled by a system of priority and overlap rules. Two coordinate systems are in use here. Images and figures are described in a user-defined coordinate system that may encompass any subset of the Cartesian plane. When an image is associated with a display in the second step of the display process, the contents of this image are mapped from user-defined coordinates into device coordinates.

In his design philosophy, Green supports the inclusion of three basic graphics primitives: text, line, and polyline. For the text primitive, a string of characters and a position indicating where this string is to be placed must be provided. Both end points of a line are required for the line primitive. Many graphics packages support the concept of a current position, thereby reducing the arguments required for specification of a graphical primitive. Unfortunately this concept becomes ambiguous when transformations are allowed in the application coordinate space. Also, most graphics displays use the idea of a cur-

rent position, but this current position rarely coincides with that of the graphics package. Because of these problems, the notion of current position is not included in this design philosophy [Green 1982a].

Green also suggests the provision of a facility whereby the user may define his own graphical primitives. In a graphics application, there are often graphical entities (shapes) that are used repeatedly throughout the program. A primitive definition facility enables the programmer to define these entities as graphical primitives. Thereafter, when an instance of one of these entities is required, it may be created through a simple declaration, just as line, text, and polyline instances are created.

Although Green's design philosophy does not discuss input handling, it is such an important aspect of any graphics system that some mention of how MacPac deals with input is necessary. MacPac maps all physical input devices into a single class which returns a coordinate position, an integer code, and a character string. When input is received the image to which that input is directed is notified and provided with the returned information. The coordinate position received by an image is always specified in the user-defined coordinate system of that image. MacPac handles the conversion from device coordinates before transmitting the input information.

The design of MacPac follows the ideas expressed in this section. In the discussion of MacPac that ensues we turn first to a description of the user's view (Chapter 4). This is elaborated by the inclusion of

display considerations (Chapter 5). A final specification of MacPac is then presented (Chapter 6). In both the description and specification of the system we make use of design languages developed by Mark Green. Before proceeding with our discussion of MacPac, an introduction to these languages and the manner in which they are used in this work is provided.

3.3 The Design Languages

In his paper "A Specification Language and Design Notation for Graphical User Interfaces" [Green 1981], Mark Green presents a methodology for the design of graphical user interfaces and a number of tools which support this methodology. The design methodology itself is in many ways user interface specific and is therefore of limited use to us in the development of MacPac. It is oriented towards the design of individual user interfaces whereas MacPac must support a wide variety of user applications and their interfaces. The tools introduced in this paper, however, provide a means of expressing design ideas that is not limited to the design of graphical user interfaces. As mentioned in Chapter 1, we will use these tools in the design of the MacPac system. In this way, we will examine their effectiveness in the design of a graphics system.

Major amongst these tools are a User Modeling Language (henceforward UML) and a Graphical User interface Specification Language (henceforward GUSL). Both of these languages depend upon a base language in which all expressions and assertions are written. An overview of these languages is presented in this chapter. More information on the structure of the languages, including detailed grammars, may be found in Appendix A. (All information on these languages is taken from [Green 1981]).

Before going on to our discussion of these languages, however, a word on the tone in which they are used in this work is required. Throughout the development of the MacPac system, these languages were also experiencing developmental changes. The paper mentioned above ([Green 1981]) provides an introduction to the languages and a simple example but does not provide information sufficient for their use in an advanced application. Further information and examples may be found in some of Green's later work (e.g. [Green 1982b]). Upon examination of these later examples, however, it becomes apparent that changes have been made to the languages since they were introduced in his 1981 paper. Unfortunately, these changes have not been published. As a result conflicting information on legal language constructs has been encountered on occasion. Because of these problems, we have used the languages as a descriptive tool in the design of MacPac. They should not be considered the basis from which a mathematical correctness analysis could be done. There are a number of loose constructs and these must be accepted as such. This does not invalidate their use in

the design of MacPac, however. The languages are useful in providing a clear and consistent design description.

3.3.1 The User Modeling Language - UML

The first step in the design of MacPac is a description of the user's view of the system. In the case of MacPac, the user is the application programmer. All objects which will be available to the user must be described. Operators must be provided to allow manipulation of these objects to suit the user's requirements. Green refers to the model constructed to represent the user's view of the system as the control model. The tool he provides to aid in creation of this model is UML.

The first major contruct in UML is the OBJECT construct. Object definitions, which are similar to type definitions, are used to describe all objects in the user's domain. There may be several instances of a given object type. Each object is described by its attributes. The type of an attribute may be either a base type (one defined in the base language), another object, or a defined theory. More information on theories will be provided later in this chapter.

The OPERATOR construct is used to describe how objects may be manipulated. An operator is defined by its pre and post conditions. The conditions themselves are assertions written in the base language. All assertions in the pre-condition part of an operator must be true for successful application of the operator. The assertions in the post-

condition section determine the effect of the operator. The operator manipulates the neccessary objects to make all post assertions true. The specific entities to be manipulated may be indicated to an operator by way of input parameters. Also, an operator may return a value. The types of these input and output parameters must be specified in the operator definition. It should be noted that the only effect an operator has is that stated explicitly by its post conditions. The state of objects not explicitly manipulated does not change. For example, if the purpose of an operator is to remove a given entity from a set, a post condition might be "NOT this_entity in set;". This operator would remove only the specified entity from the set. All other entities in the set would remain unaffected.

The UML INVARIANT construct provides a means of describing aspects of the user's view that are not readily expressed through the object and operator constructs. Each invariant is an assertion that must always be true. It is very useful in describing global characteristics of the system and giving information on the relationships between objects.

3.3.2 The Specification Language - GUSL

Once the control model is created, it may be used as the starting point for the specification of the system. The specification language developed by Mark Green, GUSL, is based on idea of state machines. Under this approach, a program is divided into a number of

state machines, each having a local state and functions capable of accessing and changing this state. Green likens a state machine to an abstract data type. In GUSL, the MODULE construct is used to represent the state machines of a system. As for objects, there may be several instances of a single module.

There are four possible components in a GUSL module. The first two deal with declaration of the entities used and manipulated by the module. The PARAMETERS component specifies the parameters required to create a module instance. These may be used to give each module instance the appropriate starting state. The DECLARATIONS component of the module may be used to declare variables for use within the module.

The third component is the DEFINITIONS component. GUSL allows the definition of syntax macros. Each macro consists of a type, the name of the macro, parameters input to the macro, and an expression that defines the macro. These macros, which are used to shorten the definitions of commonly used functions, make the contents of a module more readable.

The fourth module component is by far the largest. The FUNCTIONS component contains the function definitions of all module functions that may be used to access and/or change a module's state. There are three types of GUSL functions: V functions (VFUN), O functions (OFUN), and OV functions (OVFUN).

The first type, V functions, are used to access and represent the state of a module. Each VFUN has three sections; a function header, pre-conditions, and a function body. The function header contains the name of the function and declarations of all input parameters as well as the type of the value returned by the function. The pre-conditions section contains those assertions that must be true before the function is invoked. The function body may be either a primitive or a derived function body. A primitive function body gives the initial value of the function. The value of a primitive VFUN may be hidden, in which case it may only be accessed from within the module. A derived VFUN may be used as an interface between the current module and other modules in the specification. A derived function body contains a number of assertions. These are similar to post-conditions in that the operation of the function serves to make these assertions true. One assertion must equate a value with the function name as the value to be returned from the function invocation.

O functions are used to change the state of a module. The only way the value of a primitive VFUN may be changed is through an OFUN of the same module. No value is returned from an OFUN. Each OFUN contains pre and post assertions. The pre assertions specify the conditions under which the function may be invoked. When an OFUN is invoked, the post assertions are considered one at a time and in the order presented. The state of the module is modified for each assertion to make that assertion true. Note that, as a previous assertion may be affected by one that follows, it is possible that not all post assertions will be true after an OFUN is invoked.

The structure of an OVFUN is similar to that of an OFUN. The only difference is that an OVFUN returns a value and so must therefore, at some point in the post-conditions, equate the function name with a value.

3.3.3 The Base Language

The major components of the base language are a set of theories and a number of special forms. Green defines a theory as a data type and the operations which may be performed on any entity of this type. There are a number of pre-defined theories in the base language; integer, real, set, vector, enumeration, boolean, point, and extent. The operators of these pre-defined theories are presented in Appendix A. The base language also allows specifier-defined theories. A theory may be defined by use of a GUSL module. All operations which may be performed on this new theory must be included as functions of the module. There are two ways of referencing a theory operator (GUSL function) :

1. module_name.function_name(function_parameters)
2. argument function_name module_name

The second form is only possible if the function has a single input parameter. The argument must be of the same type as this parameter. Every object in the UML and GUSL descriptions of the system must belong to one of the pre-defined or specifier-defined theories of the base language.

As mentioned earlier all expressions and assertions of UML and GUSL are written in the base language. An expression is made up of values, theory operators, and special forms. A value may be an object, variable, parameter, or literal. An assertion is simply an expression that has a true or false value. Both UML and GUSL deal with assertions. In many cases two or more expressions are combined to create an assertion. The special forms of the base language are all assertions. There are a number of these special forms. The EXISTS and FORALL special forms provide a means of examining the objects of our system to see if at least one, respectively all, of these objects satisfies certain conditions. The LET special form allows for creation of a new entity which meets specified requirements. The IF special form lets one choose the set of assertions to be satisfied based on the value of another assertion. More information on these special forms is provided in Appendix A.

Two special characters may also be found in base language expressions. The first is the single quote ('). This character is used in conjunction with an expression to indicate use of the old, or previous, value of the expression. For example, the expression :

```
a = 'a + 1;
```

may be interpreted as "the current value of a is equal to the previous value of a plus one". The second special character is the question mark (?). This may be used in primitive VFUNs to indicate that the initial value of the VFUN is undefined. However, when a "?" is used, there must be an OFUN in the same module capable of modifying the value of that VFUN.

As we proceed with the design of MacPac, the structure and syntax of these design languages will become clearer. The constructs of UML and GUSL are fairly self-explanatory. This section has simply provided an overview of the notations we will be using in the design process.

Chapter 4

The User Model

The initial step in the design of MacPac is to describe the desired external behavior of the system. As discussed in the next two subsections, this involves determining what basic entities a user will be dealing with when using MacPac and then describing the behavior of these entities.

4.1 Objects

When using MacPac, the user is faced with a world containing a number of display devices. On each display may be seen one or more images, each of which is composed of one or more figures. As well as receiving this graphical information, the user is capable of entering information into the system via an input device. The UML OBJECTS created for these five basic entities (the world, display devices, images, figures, and input devices) are presented and discussed below.

From the user's point of view, the world may be described by the UML object :

```
OBJECT world;
   ATTRIBUTE display_set : SET OF display;
   ATTRIBUTE image_set : SET OF image;
   ATTRIBUTE figure_set : SET OF figure_transform;
   ATTRIBUTE primitive_set : SET OF primitive;
END;
```

Every display, image, figure, and graphical primitive that exists in the user's world, whether or not it can be seen or is being used at that particular point in time, is recorded in the appropriate attribute of the world object.

A display is formally described by the UML object :

```
OBJECT display;
   ATTRIBUTE display_id : integer;
   ATTRIBUTE area : extent;
   ATTRIBUTE contents : SET OF image;
   ATTRIBUTE input_group : SET OF input_device;
END;
```

The first attribute, "display_id", is an integer descriptor for the display. Each display in the user's world must have a unique "display_id" between 1 and x, where x is the number of displays contained in the "world.display_set". The next two attributes detail the size of the display ("area") and the set of images that it contains ("contents"). The "area" attribute of a display specifies the device coordinate system of that display. For an existing image to be seen on a display it must be added to the "contents" of the display. Also associated with each display is a set of input devices, "input_group".

An image is defined by the object :

```
OBJECT image;
   ATTRIBUTE contents : SET OF figure_transform;
   ATTRIBUTE positions : VECTOR OF extent;
   ATTRIBUTE window : extent;
   ATTRIBUTE chosen : boolean;
   ATTRIBUTE input_information : input_event;
END;
```

Each image is made up of a number of figure_transforms. These figure_transforms are recorded in the "contents" attribute of the image and may or may not be seen when the image is added to a display depending on the attribute "window". All figures are described in a user-defined coordinate system (UCS) which may encompass any subset of the cartesian plane. The "window" is also defined in the UCS and indicates what portion of the user's "world" will be considered when that image is displayed. Thus an image can contain much more information than is displayed at any one time and selected portions can be brought to view by changing the "window" attribute of that image. When an image is added to the "contents" of a display, an area in the coordinates of that display must be specified. This area, or viewport, controls where on the display the image will be seen. The image attribute "positions" is used to record these viewports. The ith element of "positions" contains the image's viewport on the ith display, i.e. the display whose "display_id" equals i. The value NULL is stored in the "positions" vector for all entries which correspond to displays on which the image does not appear. When an image is selected by the user, the attribute "image.chosen" becomes true and the information from the input device is stored in "input_information".

Figure_transforms are represented by the object :

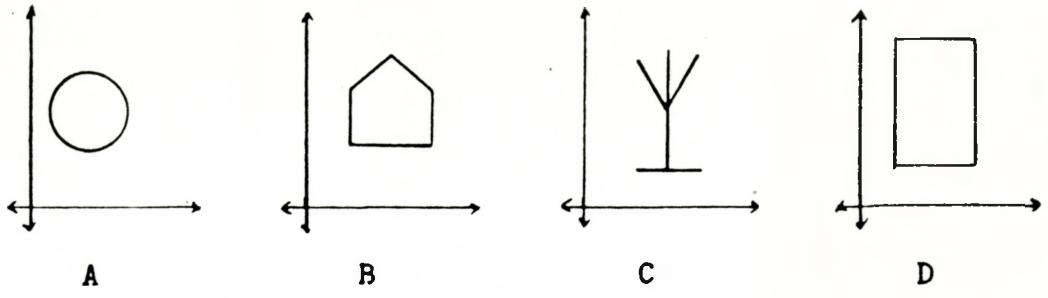
```
OBJECT figure_transform;
   ATTRIBUTE figure : element;
   ATTRIBUTE trans : transform;
END;
```

where :

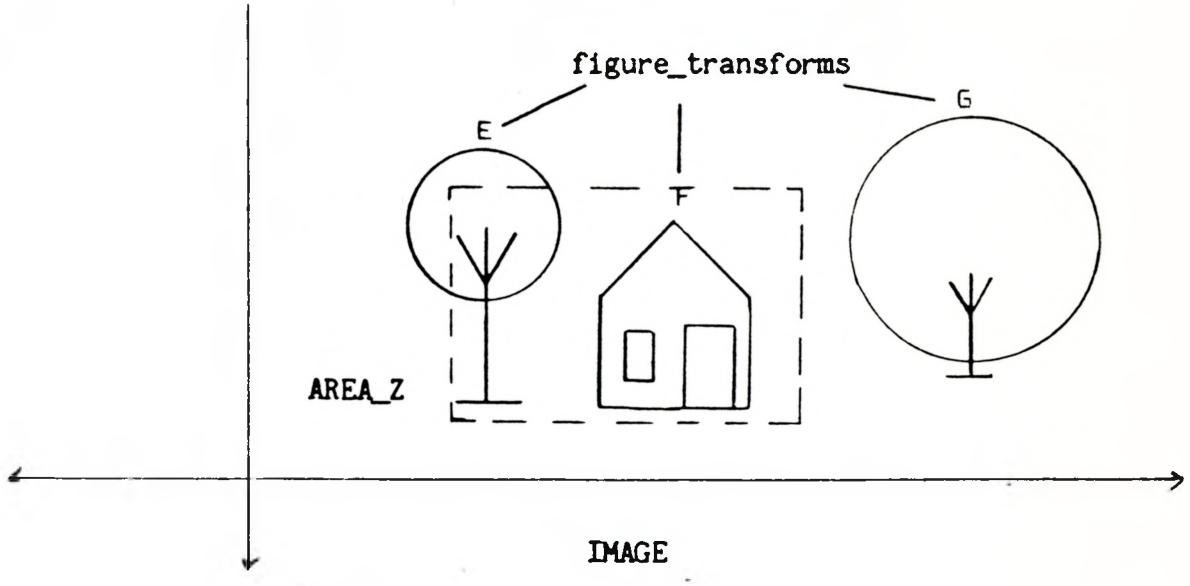
```
TYPE element = (primitive, SET OF figure_transform);
TYPE primitive = (line,text,polyline,user_defined_primitive);
```

Each figure_transform has two attributes, a "figure" and an associated transformation ("trans"). The "figure" attribute may be either a graphical primitive, in which case the figure_transform is referred to as elementary, or a collection of other figure_transforms, referred to as a composite figure_transform. Allowing one figure_transform to be defined in terms of others enables the user to define standard shapes and then use them in various figures. Although there are no formal restrictions, all the graphical information contained in a figure_transform should be logically related. The transformation associated with each figure positions that figure within a parent figure_transform or an image.

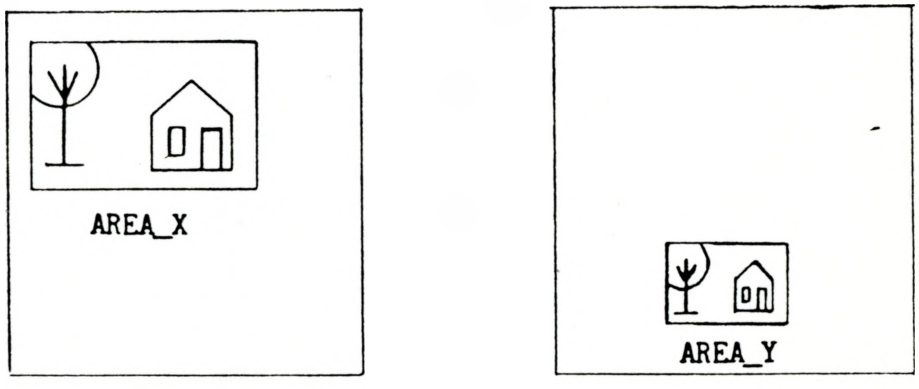
The relationship between figure_transforms, images, and displays is perhaps best illustrated by an example. Figure 4.1 presents a simple application of the MacPac system. The top of Figure 4.1 shows a number of figure_transforms which are independent of any image. Below this we see the coordinate space of an image. Three new figure_transforms have been created out of existing figure_transforms



FIGURE_TRANSFORMS



IMAGE



display_id = 1 display_id = 2

DISPLAYS

Figure 4.1 - An Example of a MacPac Application

and added to the "contents" of the image. We will examine figure_transform "E", which is meant to be a tree, in more detail. This figure_transform is composed of two child figure_transforms. The "figure" attributes of these children are figure_transforms "A" and "C". The "transform" attributes of the children position these figures together to create the tree. This tree is then positioned in the image by the "transform" of figure_transform "E". The image's "window" is denoted by the dashed rectangle, AREA_Z. The bottom of Figure 4.1 shows two displays which contain this image. The image is seen on each display in the viewport specified by the "positions" attribute of the image, i.e. the first element of "positions" would be AREA_X and the second element would be AREA_Y.

The last basic object a user encounters is the input device. For our input model all physical input devices are treated as a single object, defined by :

```
OBJECT input_device;
   ATTRIBUTE where : point;
   ATTRIBUTE code : integer;
   ATTRIBUTE string : text;
   ATTRIBUTE enabled : boolean;
   ATTRIBUTE activated : boolean;
END;
```

To be used for input an input_device must first be enabled, i.e. "enabled" must be true. While enabled, if the input_device is "activated", by the user the incoming input information will be recorded in the attributes "where", "code", and "string". The user action re-

quired for activation may be a pushing a button, hitting the enter key, or one of many other input possibilities. Any activation of the input_device while it is not enabled is ignored. The attribute "where" specifies a point in device coordinates, "code" returns an integer code, and "string" contains a character string. If the particular device in use does not provide a point, an integer code, and/or a character string, the system will provide the missing information. This object is accompanied by the object input_event :

```
OBJECT input_event;
   ATTRIBUTE where : point;
   ATTRIBUTE code : integer;
   ATTRIBUTE string : text;
   ATTRIBUTE display_id : integer;
END;
```

When an input_device is activated by the user, MacPac uses the attribute "input_device.where" to determine which image the input information should be sent to. The image selected is the image that appears on the display at this point. "Input_device.where" is then transformed into the UCS of the image and stored in "input_event.where". The input_device attributes "code" and "string" and the "display_id" of the display from which the input is received are also recorded in the input_event. This object then becomes the "input_information" for the appropriate image.

Associated with these objects are a number of assertions which must always be true. The first of these invariants is :

```
INVARIANT
  FORALL d:display | d in world.display_set
  {
    FORALL i:image | i in d.contents
    {
      EXISTS x:point | x in element(i.positions,d.display_id)
      {
        x in d.area;
      };
    };
  };
```

This specifies that each image contained in the "contents" of a display must be transformed into an area which falls at least partially within the "area" of the display. In other words, the viewport specified for an image when it is added to the "contents" of a display must overlap some portion of the "display.area". The second invariant states that the point associated with each "activated" input_device belonging to the "input_group" set of a display must fall within the "area" of the display :

```
INVARIANT
  FORALL d:display | d in world.display_set
  {
    FORALL id:input_device | id in d.input_group AND id.activated
    {
      id.where in d.area;
    };
   };
```

The above assertions must be true for each display in the "world.display_set".

Each input_device may be associated with only one display. This may be stated formally as follows :

```
INVARIANT
  FORALL d1,d2:display | d1 in world.display_set AND
                         d2 in world.display_set AND
                         d1 != d2
  {
    NOT EXISTS id:input_device | id in d1.input_group AND
                                 id in d2.input_group;
  };
```

The final invariant ensures that only one image may be chosen for input at any single point in time :

```
INVARIANT
  FORALL i1,i2:image | i1 in world.image_set AND
                       i2 in world.image_set AND
                       i1 != i2
  {
    NOT (i1.chosen AND i2.chosen);
  };
```

4.2 Operators

Having determined the basic objects a user will be dealing with when using MacPac, a description of the ways in which the user may manipulate these objects to meet his/her specific needs is required. The UML OPERATOR construct is used here to formally describe the behavior of our UML objects.

The first set of operators we will discuss are those which manipulate displays. Operators are required to control the visibility of an image as well as to change the placement of an image on a display screen. All these operators demand that the display to be manipulated be provided as an input parameter. In this way it is possible to en-

sure that the executer of the operator has access to the display device about to be changed.

The operators developed to control an image's visibility, display_image and erase_image, can be seen below :

```
OPERATOR display_image(d:display;i:image;e:extent);
   PRE
     d in world.display_set;
     i in world.image_set;
     EXISTS p:point | p in e
       {
         p in d.area;
       };
   POST
     i in d.contents;
     element(i.positions,d.display_id) = e;
END;


OPERATOR erase_image(d:display;i:image);
   PRE
     d in world.display_set;
     i in d.contents;
   POST
     NOT i in d.contents;
     element(i.positions,d.display_id) = NULL;
END;
```

The operator display_image adds a given image, i, to the "contents" of display, d. An area (or viewport), e, defined in display coordinates, must also be specified. The image will be transformed into this area for display. This viewport information is stored in the "positions" vector of the image. Erase_image simply removes a specified image from the "contents" of a given display. The viewport information held in "i.positions" for this display is also removed.

Another display manipulation operator is move_image. This operator may be used to alter the viewport of an image on a given display. The new viewport, specified in the operator parameters, must at least partially overlap the "area" of the display.

```
OPERATOR move_image(d:display;i:image;e:extent);
   PRE
     d in world.display_set;
     i in d.contents;
     EXISTS p:point | p in e
       {
         p in d.area;
       };
   POST
     element(i.positions,d.display_id) = e;
END;
```

Operators for the manipulation of images are also required. Essentially, we want to be able to create, alter, and destroy images. The image itself must be provided as input to all image alteration operators. As for displays, an image may not be manipulated without this proof of access to the object.

The UML definition of the create operator is :

```
OPERATOR create_image(w:extent) -> image;
   POST
     LET e:VECTOR OF extent | length(e) = ?;
     FORALL d:display | d in world.display_set
       {
         element(e,d.display_id) = NULL;
       };
     LET i:image | i.contents = EMPTY AND i.window = w   AND
                   i.positions = e    AND i.chosen=FALSE AND
                   i.input_information = NULL;
     create_image = i;
     create_image in world.image_set;
     NOT EXISTS d:display | d in world.display_set AND
                            create_image in d.contents;
END;
```

Once the user has decided on a particular area of interest ("w") within his world, the create_image operator may be used to return a new image with this attribute. Initially the image will not contain any figure_transforms or be seen on any display.

Once created, there are a number of ways in which one might want to alter an image. The UML definitions of the operators developed for handling these alterations are as follows :

```
OPERATOR add_to_image(i:image;f:figure_transform);
   PRE
     i in world.image_set;
     f in world.figure_set;
     NOT f in i.contents;
   POST
     f in i.contents;
END;

OPERATOR remove_from_image(i:image;f:figure_transform);
   PRE
     i in world.image_set;
     f in i.contents;
   POST
     NOT f in i.contents;
END;

OPERATOR pan_image(i:image;w:extent);
   PRE
     i in world.image_set;
   POST
     i.window = w;
END;
```

As can be seen, a necessary precondition for all alteration operators is that the image to be altered must exist - ie. it must have been previously created and therefore added to the "world.image_set". The

operators add_to_image and remove_from_image are fairly self-explanatory. They simply add or remove the specified figure_transform to or from the specified image. Note that a figure_transform can only be added to an image if it does not already exist in that image. This is in accordance with the definition of a set which does not allow more than one occurance of a member. Pan_image is useful for looking at different parts of an image. This operator allows the user to specify a new value for the image "window". As discussed in Section 4.1, the "window" field of an image specifies an area in the UCS which acts like a window onto the set of figures the image contains. Any figure, or part thereof, which falls within this "window" may be seen when the image is added to the "contents" of a display. Therefore, by changing the "window" attribute of an image the user may "pan" over different sections of the image. This operator may not affect the area of a display in which an image is seen. MacPac handles the transformation of the new "image.window" into the viewport specified for the image when it was originally added to a display.

When the user is finished with any given image, he can destroy it with the operator :

```
OPERATOR destroy_image(i:image);
   PRE
     i in world.image_set;
   POST
     FORALL d:display | d in world.display_set AND i in d.contents
       {
         erase_image(d,i);
       };
     NOT i in world.image_set;
END;
```

This operator removes the specified image from all displays of which it is a member of the "contents", and then destroys it by removing it from the "world.image_set". Note that the figure_transforms of which the image is composed are not destroyed - their relationship as members of a set is simply dissolved.

The next set of operators we will discuss are those which create, alter, and destroy figure_transforms. These operators function in similar fashion to the corresponding image manipulation operators.

The UML figure_transform creation operator is :

```
OPERATOR create_fig_trans(e:element;t:transform) -> figure_transform;
   POST
     IF NOT(e in world.primitive_set) THEN
        LET e:SET OF figure_transform | e = EMPTY;
     ENDIF;
     LET y:figure_transform | y.figure = e AND y.trans = t;
     create_fig_trans = y;
     create_fig_trans in world.figure_set;
     NOT EXISTS i:image | i in world.image_set AND
                          create_fig_trans in i.contents;
     NOT EXISTS f:figure_transform | f in world.figure_set AND
                                     create_fig_trans in f.figure;
END;
```

This operator takes as parameters an element and a transform and returns a newly created figure_transform with these attributes. If the element parameter, "e", is not a previously defined graphical primitive then it is assumed to be a dummy variable and is assigned the type SET OF figure_transform. Initially this set is empty. The element "e" then becomes the "figure" part of the figure_transform and the newly

created figure_transform is added to the "world.figure_set". At first this figure_transform may not belong to the "contents" of any image or be part of the "figure" of any parent figure_transform.

The possible alterations to a previously created figure_transform are described by the operators :

```
OPERATOR add_to_figure(f:figure_transform;new:figure_transform);
   PRE
     f in world.figure_set;
     new in world.figure_set;
     new != f;
     IF NOT (f.figure in world.primitive_set) THEN
        NOT new in f.figure;
     ENDIF;
   POST
     IF (f.figure in world.primitive_set) THEN
        LET f1:figure_transform | f1.figure = f.figure AND
                                  f1.trans = f.trans;
        LET y:SET OF figure_transform | f1 in y;
        f.figure = y;
        LET t:transform(1,0,0,1,0,0);
        f.trans = t;
     ENDIF;
     new in f.figure;
END;


OPERATOR remove_from_figure(f:figure_transform;old:figure_transform);
   PRE
     f in world.figure_set;
     NOT (f.figure in world.primitive_set);
     old in f.figure;
   POST
     NOT old in f.figure;
END;


OPERATOR transform_figure(f:figure_transform;t:transform);
   PRE
     f in world.figure_set;
   POST
     f.trans = t;
END;
```

The operator add_to_figure is used to add one figure_transform to the "figure" attribute of another. The complexity of this operator arises from the lack of restrictions on the receiving figure_transform "f" - i.e. "f" may be an elementary figure_transform. If this is the case certain changes must be made to "f" before it is possible to add "new" to "f.figure". The way the operator handles this situation is to create another figure_transform, "f1", and give it the attributes of "f". "F.figure" then becomes a SET OF figure_transforms containing "f1", and "f.trans" is set to the identity transform. The effect of this manipulation is to convert "f" to a composite figure_transform without changing the graphical information it contains. The addition of "new" to a composite "f" is very straightforward as "new" is simply added to the set "f.figure". It should be noted, however, that the add_to_figure operation is only allowed if "new" does not already exist in "f.figure". This ensures that every figure_transform within another figure_transform is uniquely identifiable and is in accordance with the definition of a set which permits only one occurance of a member. Remove_from_figure removes the figure_transform "old" from the "figure" attribute of the figure_transform "f". This operator, for obvious reasons, requires that the input figure_transform "f" be a composite figure_transform. The operator transform_figure takes a figure_transform , "f", and a transformation, "t", as parameters and assigns the value "t" to "f.trans". This enables a user to move a figure about simply by changing it's associated transform.

The operator for destroying a figure_transform is virtually identical to that for destroying an image :

```
OPERATOR destroy_fig_trans(f:figure_transform);
   PRE
     f in world.figure_set;
   POST
     FORALL f2:figure_transform | f2 in world.figure_set AND
                                  f in f2.figure
       {
         remove_from_figure(f2,f);
       };
     FORALL i:image | i in world.image_set AND f in i.contents
       {
         remove_from_image(i,f);
       };
     NOT f in world.figure_set;
END;
```

The specified figure_transform, "f", is removed from all figure_transforms and images of which it is a part and is then destroyed through removal from the "world.figure_set". However, if "f.figure" was a set of figure_transforms, the components of this set are not destroyed - they simply are no longer related to one another in the same fashion.

Another, not so obvious, operator is required for the manipulation of figure_transforms. The need for this operator is perhaps best illustrated by an example. Say that our user has defined a figure_transform called "man" which contains all the information necessary to draw a stick figure on a display. The user is also creating a figure_transform called "group". "Group.figure" is a set of figure_transforms, two of which are "man_1" and "man_2". Both of these figure_transforms have the "man" figure_transform mentioned above as their "figure" attribute and a transformation which positions the man within the larger "group"

figure as their "trans" attribute. Here is where a problem arises - what if we want one of these men to lift his arm ? If we manipulate "man_1.figure" we automatically alter "man_2.figure" as these are one and the same. A solution to this problem is a copy operator. With this operator, when several instances of a figure_transform are needed, several copies can be made, thereby enabling the user to manipulate each copy independently. The copy operator developed takes a figure_transform as input and returns a copy of this figure_transform :

```
OPERATOR copy(f:figure_transform) -> figure_transform;
   PRE
     f in world.figure_set;
   POST
     LET copy:figure_transform;
     IF f.figure in world.primitive_set THEN
        LET y:primitive | y = f.figure;
        y in world.primitive_set;
        copy.figure = y;
     ELSE
        FORALL x:figure_transform | x in f.figure
          {
            copy(x) in copy.figure;
          };
     ENDIF;
     copy.trans = f.trans;
     copy in world.figure_set;
     NOT EXISTS i:image | i in world.image_set AND
                          copy in i.contents;
END;
```

The final operator, select_image, enables the user to send input information to an image. Whenever an enabled input_device is activated, this operator is invoked with the input_device as argument. Select_image examines the display associated with this input_device. If an image appears on the display at the point specified by the in-

put_device then the input information is relayed to this image and the image is flagged as chosen. If the point specified by the input_device does not fall on any visible image then no action is taken.

```
OPERATOR select_image(id:input_device);
   PRE
     id.enabled = TRUE;
     id.activated = TRUE;
     EXISTS d:display | d in world.display_set
       {
         id in d.input_group;
         id.where in d.area;
       };
   POST
     EXISTS d:display | d in world.display AND
                        id in d.input_group
       {
         LET b:boolean |
             b = EXISTS i:image | i in d.contents
                   {
                     id.where in element(i.positions,d.display_id);
                   };
         IF b THEN
           EXISTS i:image | i in d.contents AND
                    id.where in element(i.positions,d.display_id))
             {
               LET t:transform |
               t.apply(i.window) = element(i.positions,d.display_id);
               i.input_information.where = t.rapply(id.where);
               i.input_information.code = id.code;
               i.input_information.string = id.string;
               i.input_information.display_id = d.display_id;
               i.chosen = TRUE;
             };
         ENDIF;
       };
END;
```

So far we have discussed the objects a user may encounter when using MacPac and several operations which may be performed on these objects. Nothing has been said, however, about how figure_transforms and images are actually seen on a display. The following section discusses this area in detail, modifying the user model as necessary.

Chapter 5

Display Considerations

When one considers how the actual display will be handled in a system using MacPac, several questions come to mind. For instance, what does the user actually see on the display screen and how will this information be represented? What happens if two images overlap on a display - does one have priority over the other? Also, we have discussed several operators which alter images and figure_transforms but nothing has been said about when the alterations become visible to the user.

In this chapter we will address these display considerations. The user view presented in Chapter 4 is used as starting point from which to build a new model; one which incorporates the information necessary to describe what the user sees. We first consider the objects of MacPac. Several new attributes are added to hold information relevant to their visual display. The operators which manipulate these objects are then examined. Their function is expanded to include the effect of the operator on the visual representation of our MacPac objects. For reference purposes, a compendium of these operators is provided in Appendix C.

5.1 Objects

Before other display related problems can be considered, one needs an answer to the very important question - what does the user actually see on the display screen and how will this information be represented? As the majority of graphical display devices in use today are raster based it seems logical to try to describe what is seen on a display in a form compatible with this. Fortunately, it turns out that "all the important properties of a raster display can be represented by a GUSL module" [Green,1982b,p.7] which we call raster_display. As discussed in Chapter 3, any GUSL module may be used as the type of an attribute. Therefore, our problem of display representation is solved simply by adding a new attribute, "screen", of type raster_display, to the display object. With this representation it is possible to describe what the user sees on a display at any point in time as well as how what the user sees is affected by the various operators.

An attribute of type raster_display was also added to the image object. As well as providing information on what each image looks like in the absence of a display screen and other images, this new attribute, called "picture", facilitates the display_image operation in that the information in "image.picture" is in the same form as the information in "display.screen". Therefore, when adding new image information to a "display.screen", no conversion, other than from UCS to device coordinates, is required.

To handle the problem of image overlap it is necessary to impose an ordering on the images contained in a display. This was accomplished by changing the type of the attribute "display.contents" from a "SET OF image" to a "VECTOR OF image". The larger the vector index of an image the higher the "priority" of that image - in other words, when two or more images overlap on a display screen, the image with the largest vector index is seen in the overlap area. It is important to realize that this ordering is display, NOT image, dependent. An image does not have any kind of "priority" associated with it in the absence of a display and two different displays may contain the same image at very different vector indices. The problems of figure_transform overlap within an image and within a parent figure_transform are handled in much the same way. The type of the attribute "image.contents" was changed to "VECTOR OF figure_transform" and at any point in an image where two or more figure_transforms overlap it is the figure_transform with the largest vector index which is seen. The definition of element, the type of the attribute "figure_transform.figure", was changed from :

```
TYPE element = (primitive,SET OF figure_transform);
```

to :

```
TYPE element = (primitive,VECTOR OF figure_transform);
```

thereby establishing an ordering amongst figure_transforms within a figure_transform which allows us to handle this overlap problem in the same way - by relative vector indices.

The problem of when new information should become visible to the user is encountered when the user makes a change to something he is currently viewing on a display. Should the change appear as soon as it is made or should the user have to explicitly request an update, thereby allowing him to make several changes before any are seen? When one looks at possible uses for the system, it soon becomes obvious that both updating alternatives are desirable; the one chosen for use is very situation dependent. In response to this, the boolean attribute "instant_update" was added to the world object. This attribute allows the user to choose the updating alternative most suited to his specific purposes. When "world.instant_update" is true any alterations made to figure_transforms and images on display are seen immediately. When "world.instant_update" is false these changes will not appear until the affected displays are refreshed. A new operator, refresh_display, developed for this purpose will be introduced later in this chapter.

The addition of this user option, however, creates some display related problems of its own. For instance, if "world.instant_update" is false and we decide to make changes to an image on display how will these changes be remembered until they are used. If we go ahead and make the changes to the image but not the display screen(s), how will we keep track of whether or not the display screen reflects the "contents" of the display. Our solution to this problem involved the addition of the boolean attributes "up_to_date" and "altered" to the display and image objects respectively. Whenever changes are made to an image the changes are immediately recorded in the appropriate image

attribute ("window", "positions", or "contents") and "image.altered" is set to true. When the "picture" attribute is updated to reflect these changes, "image_altered" is reset to false. If "world.instant_update" is false, all displays which contain this image are flagged as not up_to_date as the display "screen" may no longer match the contents of the display.

Another feature of any visual display is colour information and control. Both the display and image objects have been extended to accomodate colour handling. The attribute "colour_table" was added to the display object. This attribute is of type colour_map. As for raster_display, this entity may been represented by a GUSL module and so may therefore be used as the type of an attribute. Basically, a colour_map is a an array of bytes which are interpreted by the display device to produce colours. When dealing with colour information in an application, integral values are used. These values are actually indices into the colour_map, from which the byte pattern used to produce the colour is retrieved. Three new colour attributes have been added to the image object : "colour_start", "colour_range", and "background". "Colour_start" and "colour_range" are used to specify the portion of a display's "colour_table" which the image may access. All colours in the image's "picture" must fall within 0 and "colour_range". If the colour of a figure_transform contained in the image is greater than "colour_range", the figure is displayed as colour 0. When the image is added to a display, "colour_start" indicates where in the display's "colour_table" to start retrieving colours from. For example, if

"colour_start" is 25 and a pixel in the images "picture" is colour 10, when this image is added to a display the corresponding "screen" pixel will be set to colour 35. Again, if the calculated value for a "screen" pixel is greater than the display's "colour_table" maximum, the pixel is set to colour 0. The attribute "background" allows the user to specify a background colour for each of his images.

The last two attributes added to the image object are "fill_info_points" and "fill_info_colours". These attributes enable the user to specify areas in the image's UCS which are to filled and the colours which are to be used to fill them. "Fill_info_points" is a vector of points and "fill_info_colours" is a vector of colours. These vectors function as a single unit. The ith entry of fill_info_colours" specifies the colour to use in filling the area of which the ith entry of "fill_info_points" is an interior point.

With the addition of these new attributes, the object definitions appear as follows :

```
OBJECT world;
   ATTRIBUTE display_set : SET OF display;
   ATTRIBUTE image_set : SET OF image;
   ATTRIBUTE figure_set : SET OF figure_transform;
   ATTRIBUTE primitive_set : SET OF primitive;
   ATTRIBUTE instant_update : boolean;
END;
```

```
OBJECT display;
   ATTRIBUTE display_id : integer;
   ATTRIBUTE area : extent;
   ATTRIBUTE contents : VECTOR OF image;
   ATTRIBUTE screen : raster_display;
   ATTRIBUTE up_to_date : boolean;
   ATTRIBUTE input_group : SET OF input_device;
   ATTRIBUTE colour_table : colour_map;
END;


OBJECT image;
   ATTRIBUTE contents : VECTOR OF figure_transform;
   ATTRIBUTE positions : VECTOR OF extent;
   ATTRIBUTE window : extent;
   ATTRIBUTE picture : raster_display;
   ATTRIBUTE altered : boolean;
   ATTRIBUTE background : integer;
   ATTRIBUTE colour_start : integer;
   ATTRIBUTE colour_range : integer;
   ATTRIBUTE fill_info_points : VECTOR OF point;
   ATTRIBUTE fill_info_colours : VECTOR OF integer;
   ATTRIBUTE chosen : boolean;
   ATTRIBUTE input_information : input_event;
END;
```

The figure_transform, input_device, and input_event objects were unchanged by these display considerations.

Before going on to examine the effect of these display considerations on our MacPac operators, it is necessary to introduce some functions which are required in the next section but are not presented in detail until Chapter 6. These functions, which operate on the objects and modules defined for MacPac, are similar to those mentioned in Chapter 3 that we have been using to operate on the predefined theories of Mark Green's design language. Firstly, all objects representing graphical entities have the two functions 'in' and 'colour'. Both of these functions take a point as input. Function 'in' returns true if

that point is in the graphical entity. Function 'colour' returns an integer which represents the colour of the graphical entity at that point. The objects representing graphical entities in MacPac are figure_transform, line, polyline, text, and user_defined_primitives. Another object, or module, which is used in MacPac is the raster_display. Two functions which operate on entities of this type are 'set_pix' and 'pixel'. Function 'set_pix' is used to assign a colour to a specified pixel in a raster_display. Function 'pixel' takes a point as parameter and returns the colour of the pixel at that point in the raster_display. The last set of functions which require introduction here are those which may be applied to transforms. Function 'apply' takes either a point or an extent as its only parameter. The transformation which this module represents is applied to the point, or all points in the extent, and the transformed point or area is returned. The inverse function is 'rapply'. This function applies the inverse of the transformation to an input point, returning the point which results from this operation.

5.2 Operators

Now that a representation has been developed to describe what individual images and displays look like, the operators of Chapter 4 can be altered to include the effect they have on this representation. It soon becomes evident that many of the operators require similar information on the current structure of the user's world and perform similar modifications to objects in this world. To reduce the resul-

ting redundancy in the base language "code" of the operators, several functions are defined externally for use by more than one operator. It should be noted, however, that these functions may not be invoked from outside of an operator. With this restriction we need not be concerned with pre-conditions for their use or the order in which they are invoked as this is under the control of the calling operator. These functions are presented and discussed below along with the operators which make use of them.

In handling the overlap problems discussed in the previous section, two basic functions - upper_image and upper_figure - prove useful in several of the operators. Upper_image takes in a display, an image, and a point on the display which the image maps to. This function returns true if the given image has the highest vector index of all images in the display's "contents" which also map to the given point. Upper_figure behaves in much the same fashion in determining the upper figure_transform in an image or a parent figure_transform at a particular point. The input parameters to this function are a vector ("v") of figure_transforms (to accomodate both an "image.contents" and a "figure_transform.figure"), a figure_transform ("f"), and a point ("p") in the image or parent figure_transform which the figure_transform "f" maps to. Again, true is returned if the input figure_transform "f" has the highest vector index of all figure_transforms in "v" which also map to the point "p".

```
upper_image(d:display;i:image;p:point) -> boolean;
  {
    LET int1:integer ¦ i = element(d.contents,int1);
    upper =
      FORALL i2:image ¦ i2 in d.contents AND i2 != i AND
                        p in element(i2.positions,d.display_id)
        {
          LET int2:integer ¦ i2 = element(d.contents,int2);
          int2 < int1;
        };
  };


upper_figure(v:VECTOR OF figure_transform,
                      f:figure_transform,p:point)->boolean;
  {
    LET int1:integer ¦ f = element(v,int1);
    upper =
      FORALL f2:figure_transform ¦ f2 in v AND f2 != f AND
                                   f2.trans.rapply(p) in f2
        {
          LET int2: integer ¦ f2 = element(v,int2);
          int2 < int1;
        };
  };
```

As mentioned in the previous section, when an image is altered, all new information is stored in the appropriate image attributes ("window", "positions", "contents", etc). At this time the raster_display "image.picture", from which the image is displayed, must also be updated. The function update_image_picture proves very useful here. This function completely recreates the "picture" attribute of an image from information stored in other image attributes. The "picture" raster_display is redefined from the "window" of the image and then each pixel of "picture" is set according to the figure_transforms stored in the "contents" of the image. Finally, all area fill requests which affect the image "picture" are executed.

```
update_image_picture(i:image);
  {
    LET r:raster_display(truncate(i.window.lower_left),
                         truncate(i.window.upper_right));
    i.picture = r;
    FORALL p:point | p in i.picture
      {
        i.picture.set_pix(p,i.background);
      };
    FORALL f:figure_transform | f in i.contents
      {
        FORALL figpt:point | figpt in f AND
                             f.trans.apply(figpt) in i.picture
          {
            IF upper_figure(i.contents,f,f.trans.apply(figpt)) THEN
              IF f.colour(figpt) > i.colour_range THEN
                i.picture.set_pix(f.trans.apply(figpt),0);
              ELSE
                i.picture.set_pix(f.trans.apply(figpt),f.color(figpt));
              ENDIF;
            ENDIF;
          };
      };
    update_fill(i);
    i.altered = FALSE;
  };
```

The two functions, update_fill and invoke_fill, handle all area fill operations. Update_fill examines each point of the given image's "fill_info_points" vector. If the point falls within the "picture" of the image then the invoke_fill function is executed. Update_fill determines the existing ('old') colour of the image at the fill point specified. The 'new' fill colour is retrieved from the "fill_info_colours" vector. If this colour is greater than the image's "colour_range", the 'new' fill colour is set to 0. The 'old' and 'new' colours are sent to the invoke_fill function along with the point at which to start filling. In the invoke_fill function, the input point is examined to see if the colour at this point is the 'old' colour. If

so, the colour of the pixel at this point is changed to the 'new' colour, and the eight points which neighbor the input point are examined via a recursive call to invoke_fill. The algorithm in use here is a flood-fill of an interior-defined 8-connected region.

```
update_fill(i:image);
  {
    FORALL int:integer | 0 < int <= length(i.fill_info_points)
      {
       LET p:point | p = truncate(element(i.fill_info_points,int));
       IF p in i.picture THEN
          LET new_colour:integer |
              new_colour = element(i.fill_info_colours,int);
          IF new_colour > i.colour_range THEN
             new_colour = 0;
          ENDIF;
          LET old_colour:integer | old_colour = i.picture.pixel(p);
          invoke_fill(i,p,old_colour,new_colour);
       ENDIF;
      };
  };

invoke_fill(i:image;p:point;old,new:integer);
  {
    IF p in i.picture THEN
       IF i.picture.pixel(p) = old THEN
          i.picture.set_pix(p,new);
          LET p1:point(p.x,p.y+1);
          invoke_fill(i,p1,old,new);
          LET p2:point(p.x,p.y-1);
          invoke_fill(i,p2,old,new);
          LET p3:point(p.x+1,p.y);
          invoke_fill(i,p3,old,new);
          LET p4:point(p.x+1,p.y+1);
          invoke_fill(i,p4,old,new);
          LET p5:point(p.x+1,p.y-1);
          invoke_fill(i,p5,old,new);
          LET p6:point(p.x-1,p.y);
          invoke_fill(i,p6,old,new);
          LET p7:point(p.x-1,p.y+1);
          invoke_fill(i,p7,old,new);
          LET p8:point(p.x-1,p.y-1);
          invoke_fill(i,p8,old,new);
       ENDIF;
    ENDIF;
  };
```

If "world.instant_update" is true then every time we update the "picture" of an image we must also update all displays which contain that image. The function update_displays was developed to handle this operation. Update_displays takes in an image, "i", and then updates this image on the "display.screen" of each display in the "world.display_set" which contains "i". This update makes use of the current "picture" and "positions" attributes of the image and takes into account the position of the image in the "contents" of each display.

```
update_displays(i:image);
  {
    FORALL d:display | d in world.display_set AND i in d.contents
      {
        LET t:transform |
            t.apply(i.window) = element(i.positions,d.display_id);
        FORALL p:point | p in d.screen AND
                         p in element(i.positions,d.display_id)
          {
            IF upper_image(d,i,p) THEN
               LET c:integer | c = i.picture.pixel(t.rapply(p)) +
                                   i.colour_start;
               IF c > d.colour_table.max THEN
                  c = 0;
               ENDIF;
               d.screen.set_pix(p,c);
            ENDIF;
          };
      };
  };
```

Another function used in several of the operators is shrinkvector. This function takes in a vector of images or figure_transforms, "v", and an image or figure_transform, "item", which is an element of

this vector. "Item" is removed from the vector "v" and the vector is compacted, i.e. all elements of the vector with a higher index than "item" are shifted to the left by one (their vector index is decreased by one). In this way any holes which removal of an element may create in a vector are eliminated. This function is useful for removing an image from the "contents" of a display or a figure_transform from the "contents" of an image or the "figure" of a parent figure_transform.

```
shrinkvector(v:(VECTOR OF image,VECTOR OF figure_transform);
             item:(image,figure_transform));
  {
    LET index:integer ¦ item = element(v,index);
    FORALL index2:integer ¦ index <= index2 < length(v)
      {
        element(v,index2) = element('v,index2+1);
      };
    length(v) = length('v) - 1;
  };
```

With the use of these functions it is now a simple matter to modify the MacPac operators of Chapter 4 to include the effects they have on the image and display attributes which control what the user sees. Before going on to discuss these changes, however, a new operator, refresh_display, is introduced. This operator completely recreates the "screen" attribute of a specified display from the information contained in the "contents" and "area" of the display. The "screen" is initialized based on the "area" of the display and all "screen" pixels are set to colour 0. Each image in the "contents" of the display is then examined and all pixels of the display screen which are affected by the image are set to the appropriate colour. A major

purpose of this operator is to provide the user with a mechanism whereby he may explicitly request a refresh of a display screen when "world.instant_update" is false. Refresh_display will update the display "screen" to reflect all changes made to images in "display.contents" since the last refresh. As this operator brings a display "screen" up-to-date with its "contents", the flag "up_to_date" is (re)set to true.

```
OPERATOR refresh_display(d:display);

  PRE
    d in world.display_set

  POST
    IF  NOT d.up_to_date  THEN
      LET s:raster_display(truncate(d.area.lower_left),
                           truncate(d.area.upper_right));
      d.screen = s;
      FORALL p:point | p in d.screen
        {
          d.screen.set_pix(p,0);
        };
      FORALL i:image | i in d.contents
        {
          LET t:transform |
              t.apply(i.window) = element(i.positions,d.display_id);
          FORALL p:point | p in d.screen AND
                           p in element(i.positions,d.display_id)
            {
              IF upper_image(d,i,p) THEN
                 LET c:integer | c = i.picture.pixel(t.rapply(p)) +
                                     i.colour_start;
                 IF c > d.colour_table.max THEN
                    c = 0;
                 ENDIF;
                 d.screen.set_pix(p,c);
              ENDIF;
            };
        };
      d.up_to_date = TRUE;
    ENDIF;
END;
```

The first two operators requiring modification are those which control an image's visibility - display_image and erase_image. The new display_image operator, seen below, serves two purposes. If the specified image, "i", does not already exist in the "contents" of the given display, "d", then this image is added to the end of the "d.contents" vector. The input viewport, "e" is recorded in the "positions" attribute of the image for use when updating the display "screen". If "i" is already an element of "d.contents" it is removed from this vector by a call to shrinkvector. Display_image then adds "i" to the end of "d.contents", thereby ensuring that this image has highest priority when it comes to overlap considerations. Note that display_image may not be used to move an image on a display "screen". If the image already exists in the "contents" of the display then the viewport, "e", is set to the value currently held in "i.positions" for this display. The value for "e" sent as input to this operator is overwritten. If "world.instant_update" is true, the display "screen" is updated to reflect this new information. The end result is that, when the display is updated, the image "i" will be seen above all others in the portion of the display "screen" specified by the viewport area "e".

```
OPERATOR display_image(d:display;i:image;e:extent);
  PRE
    d in world.display_set;
    i in world.image_set;
    IF i in d.contents THEN
       e = element(i.positions,d.display_id);
    ENDIF;
    EXISTS p:point | p in e
      {
        p in d.screen;
      };
```

```
  POST
    d.up_to_date = FALSE;
    IF NOT (element(d.contents,length(d.contents)) = i) THEN
       IF i in d.contents THEN
          shrinkvector(d.contents,i);
       ENDIF;
       element(d.contents,length(d.contents)+1) = i;
    ENDIF;
    element(i.positions,d.display_id) = e;
    IF world.instant_update THEN
       LET t:transform | t.apply(i.window) = e;
       FORALL p:point | p in d.screen AND p in e
         {
           LET c:integer | c = i.picture.pixel(t.rapply(p)) +
                               i.colour_start;
           IF c > d.colour_table.max THEN
              c = 0;
           ENDIF;
           d.screen.set_pix(p,c);
         };
       d.up_to_date = TRUE;
    ENDIF;
END;
```

The erase_image operator is used to erase a given image "i" from the "screen" of a specified display "d". Erase_image calls the function shrinkvector to remove "i" from "d.contents" and then resets the image's viewport for this display to NULL. If "world.instant_update" is false then no further action need be taken. If "world.instant_update" is true, a call to refresh_display updates the display "screen" from its "contents" which, since it no longer contains the image, has the effect of removing "i" from the screen.

```
OPERATOR erase_image(d:display;i:image);
  PRE
    d in world.display_set;
    i in d.contents;
```

```
  POST
    d.up_to_date = FALSE;
    shrinkvector(d.contents,i);
    /* NOT i in d.contents */
    element(i.positions,d.display_id) = NULL;
    IF world.instant_update THEN
      refresh_display(d);
    ENDIF;
END;
```

The move_image operator allows the user to move an image which appears on a display to a different area on the "screen" of that display. The new viewport in which the image is to appear must be provided as an input parameter to the operator. This new viewport information is stored in the "positions" attribute of the image. If world.instant_update is true, refresh_display is called to handle the repositioning of the image on the display "screen". It should be noted that this operator does not change the priority of the image in the "contents" of the display. Therefore, when the image is displayed in its new viewport, parts of it may be obscured by images of higher priority.

```
OPERATOR move_image(d:display;i:image;e:extent);
  PRE
    d in world.display_set;
    i in d.contents;
    EXISTS p:point | p in e
      {
        p in d.screen;
      };
  POST
    d.up_to_date = FALSE;
    element(i.positions,d.display_id) = e;
    IF world.instant_update THEN
       refresh_display(d);
    ENDIF;
END;
```

The image manipulation operators also require modification to reflect these display considerations. The first of these operators to be considered is the image creation operator. This operator has been extended to initialize the new image attributes added in section 5.1 as well as the "contents" attribute whose type was changed from SET OF to VECTOR OF figure_transform. As can be seen, three new input parameters have been added to the operator, allowing the colour information of the image to be specified by the user when the image is created. The "picture" and "altered" attributes of the image are initialized in the function update_image_picture. As the "contents" of the image is empty at this point, all pixels of the "picture" raster_display, defined from the "image.window", are set to the specified background color.

```
OPERATOR create_image(w:extent;bkgd,c_start,c_range:integer) -> image;
  POST
    LET pos_init:VECTOR OF extent | length(pos_init) = ?;
    FORALL d:display | d in world.display_set
      {
        element(pos_init,d.display_id) = NULL;
      };
    LET con_init:VECTOR OF figure_transform | length(con_init) = 0;
    LET fill_p_init:VECTOR OF point | length(fill_p_init) = 0;
    LET fill_c_init:VECTOR OF integer | length(fill_c_init) = 0;
    LET i:image |
        i.contents = con_init AND i.window = w AND
        i.positions = pos_init AND  i.chosen = FALSE AND
        i.input_information = NULL AND i.background = bkgd AND
        i.colour_start = c_start AND i.colour_range = c_range AND
        i.fill_info_points = fill_p_init AND
        i.fill_info_colours = fill_c_init;
    create_image = i;
    update_image_picture(create_image);
    create_image in world.image_set;
    NOT EXISTS d:display | d in world.display_set
          {
            create_image in d.contents;
          };
END;
```

The next set of operators to be modified are the image alteration operators: add_to_image, remove_from_image, and pan_image. As discussed in the previous section all changes made to an image are immediately recorded in the relevant attribute - "window", "position", or "contents" - of the image. The "picture" attribute of the image must then be updated to reflect these changes. The "image.altered" attribute acts as a flag throughout this process. When the "window", "position", or "contents" of an image is altered the flag is set to true. When the "picture" of the image is updated, thereby making all image information consistent, "image.altered" is set back to false. If "world.instant_update" is true, all displays containing the image being altered must be updated to reflect the changes. Otherwise, the "up_to_date" flag of these displays must be set to false. It should be noted that, although these operators may change the appearance of an image on a display "screen", they can not change where the image is seen or the priority of an image on a display. The alteration process outlined here accounts for the basic structure of all three alteration operators, discussed in more detail below.

The add_to_image operator adds a given figure_transform, "f", to a given image, "i", simply by adding "f" to the end of the vector of figure_transforms "i.contents". The new figure must also be added to the "picture" of the image. This is handled by the function update_image_picture which recreates the "picture" of the image from its

"contents", of which "f" is now a member. If "world.instant_update" is true, the call to update_displays updates the "display.screen", with respect to the image "i", of all displays that contain "i".

```
OPERATOR add_to_image(i:image;f:figure_transform);
  PRE
    i in world.image_set;
    f in world.figure_set;
    NOT f in i.contents;
  POST
    element(i.contents,length(i.contents)+1) = f;
    /* f in i.contents */
    i.altered = TRUE;
    update_image_picture(i);
    IF world.instant_update THEN
       update_displays(i);
    ELSE
       FORALL d:display | d in world.display_set AND i in d.contents
         {
           d.up_to_date = FALSE;
         };
    ENDIF;
END;
```

The remove_from_image operator performs the reverse of the add_to_image operation. A specified figure_transform, "f", is removed from a given image, "i", by removing "f" from the vector "i.contents". Again, the "picture" of the image and, if "world.instant_update" is true, the "screen" of each associated display must be updated. As in the add_to_image operator, the functions update_image_picture and update_displays handle this procedure.

```
OPERATOR remove_from_image(i:image;f:figure_transform);
  PRE
     i in world.image_set;
     f in i.contents;
```

```
  POST
    shrinkvector(i.contents,f);
    /* NOT f in i.contents */
    i.altered = TRUE;
    update_image_picture(i);
    IF world.instant_update THEN
       update_displays(i);
    ELSE
       FORALL d:display | d in world.display_set AND i in d.contents
         {
           d.up_to_date = FALSE;
         };
    ENDIF;
END;
```

Pan_image allows the user to change the "window" attribute of a given image. In this way, the user may alter the window used to view the "contents" of an image. The function update_image_picture is called to redefine the "picture" attribute of the image from its new "window". This function also updates the pixels of the new "picture" from the information in "image.contents". If "world.instant_update" is true, the displays containing this image are updated with the new "picture" of the image. Again, this operator does not change the position or priority of an image on any display. Only the appearance of the image may change.

```
OPERATOR pan_image(i:image;w:extent);

  PRE
    i in world.image_set;

  POST
    i.window = w;
    i.altered = TRUE;
    update_image_picture(i);
```

```
      IF world.instant_update THEN
         update_displays(i);
      ELSE
         FORALL d:display | d in world.display_set AND i in d.contents
           {
             d.up_to_date = FALSE;
           };
      ENDIF;
   END;
```

Two new operators are required to manipulate the new image attributes "fill_info_points" and "fill_info_colours". The user should be able to add and remove fill information as the contents and uses of an image change. The operators add_fill_info and remove_fill_info, seen below, provide this facility.

Add_fill_info may be used to add a new area to be filled. This area is specified by providing a point within the area to the add_fill_info operator. The 'new' colour the area is to be filled with must also be specified. The input point and colour are added to corresponding locations in the "fill_info_points" and "fill_info_colours" vectors. If the point designated is within the "picture" of the image, the operator performs a flood-fill on the "picture" attribute. This involves changing all pixels connected to the specified point and of the same original colour to the 'new' colour. The function invoke_fill is used to handle this operation. Note that there is no restriction on the input point, i.e. it does not have to fall within the "picture" of the image. In this way fill information may exist for all areas of the image. The actual fill operation will be executed when the affected portion of the image falls within the image "window". There is also no

restriction that the fill colour be within the image's "colour_range". However, if the specified colour is outside of this "colour_range", colour 0 will be used for filling the indicated area of the image "picture". Add_fill_info returns an integer which represents the index of the new fill information in the "fill_info_points" and "fill_info_colours" vectors.

```
OPERATOR add_fill_info(i:image;p:point;c:integer) -> integer;
  PRE
    i in world.image_set;
    c >=  0;
  POST
    LET next:integer |
        next = length(i.fill_info_points) + 1;
    element(i.fill_info_points,next) = p;
    element(i.fill_info_colours,next) = c;
    add_fill_info = next;
    IF p in i.picture THEN
       LET intp:point | intp = truncate(p);
       LET new_colour:integer | new_colour = c;
       IF new_colour > i.colour_range THEN
          new_colour = 0;
       ENDIF;
       LET old_colour:integer | old_colour = i.picture.pixel(p);
       invoke_fill(i,intp,old_colour,new_colour);
       IF world.instant_update THEN
          update_displays(i);
       ELSE
          FORALL d:display | d in world.display_set AND
                             i in d.contents
            {
              d.up_to_date = FALSE;
            };
       ENDIF;
    ENDIF;
END;
```

The operator remove_fill_info may be used to remove fill from an area of a given image. This operator must be provided with the index of the fill information. The entries of the "fill_info_points" and

"fill_info_colours" vectors at this index are removed and the vectors are compressed. The image "picture" is then recreated from scratch via a call to update_image_picture. As the removed fill information no longer exists in the image attributes, the area will no longer be filled in the image "picture". For both operators, if "world.instant_update" is true, all displays containing the altered image must be updated. Update_displays is called to handle this function.

```
OPERATOR remove_fill_info(i:image;index:integer);
  PRE
    i in world.image_set;
    index <= length(i.fill_info_points);
  POST
    FORALL int:integer | index <= int < length(i.fill_info_points)
      {
        element(i.fill_info_points,int) =
                              element('i.fill_info_points,int+1);
        element(i.fill_info_colours,int) =
                              element('i.fill_info_colours,int+1);
      };
    length(i.fill_info_points) = length('i.fill_info_points) - 1;
    length(i.fill_info_colours) = length('i.fill_info_colours) - 1;
    update_image_picture(i);
    IF world.instant_update THEN
       update_displays(i);
    ELSE
       FORALL d:display | d in world.display_set AND i in d.contents
         {
           d.up_to_date = FALSE;
         };
    ENDIF;
END;
```

The next set of operators to be considered are the figure_transform manipulation operators. As any change to a figure_transform may affect an image on display, these operators must be extended to include the alterations to the figure_transform and associated image and

display objects necessary to control what the user sees. Before going on to discuss the figure_transform operators, however, a function which proved very useful in the modification of these operators needs to be defined and discussed.

This function, update_images, takes in a figure_transform, "f", and then completely updates the "image.picture" of all images which contain this figure_transform. In this way any changes which have been made to the input figure_transform will now be accounted for in the "picture" of each image. If "world.instant_update" is true, these changes are relayed to the necessary displays by a call to update_displays for each newly updated image. If "world.instant_update" is false, all displays containing these images are flagged as not up-to-date as the display "screen" may no longer match the "contents" of the display. Due to the possible structure of a figure_transform, however, the images containing "f" may not be the only images effected by the changes to "f". It may happen that "f" is a member of the "figure" of another figure_transform which in turn constitutes part of the "figure" of several other figure_transforms, etc.. At each level the parent figure_transform may belong to the "contents" of one or more images, which may or may not be on display. The changes in these figure_transforms, due to the changes in "f", must be relayed to these images and displays. Update_images handles this situation by calling itself recursively for every figure_transform which contains the current input figure_transform.

```
update_images(f:figure_transform);
  {
    FORALL i:image | i in world.image_set AND f in i.contents
      {
        update_image_picture(i);
        IF world.instant_update THEN
           update_displays(i);
        ELSE
           FORALL d:display | d in world.display_set AND
                              i in d.contents
             {
               d.up_to_date = FALSE;
             };
        ENDIF;
      };
    FORALL f2:figure_transform | f2 in world.figure_set AND
                                 f in f2.figure
      {
        update_images(f2);
      };
  };
```

The first figure_transform manipulation operator to be considered is create_fig_trans. As this operator simply creates a new figure_transform, unassociated initially with any image, very few changes in its original structure were required to accomodate display considerations. Create_fig_trans was modified, as were all the figure_transform manipulation operators, to take into account the change from "SET OF figure_transform" to "VECTOR OF figure_transform" in the type definition of element.

```
OPERATOR create_fig_trans(e:element;t:transform) -> figure_transform;
  POST
    IF NOT(e in world.primitive_set) THEN
       LET e:VECTOR OF figure_transform | length(e) = 0;
    ENDIF;
    LET y:figure_transform | y.figure = e AND y.trans = t;
    create_fig_trans = y;
    create_fig_trans in world.figure_set;
```

```
    NOT EXISTS i:image | i in world.image_set AND
                         create_fig_trans in i.contents;
    NOT EXISTS f:figure_transform | f in world.figure_set AND
                                    create_fig_trans in f.figure;
END;
```

The modified figure_transform manipulation operators are shown below. As can be seen, in spite of the different alterations to the input figure_transform "f" which each operator makes, they all exert their effect on what the user sees in exactly the same way. In add_to_figure the "new" input figure_transform is added to the end of the vector "f.figure". Remove_from_figure removes the figure_transform "old" from the "figure" attribute of "f" through a call to shrinkvector. The transform_figure operator simply assigns the new input transformation to the "trans" attribute of the specified figure_transform "f". In each operator, after the appropriate alterations to "f" have been made, the changes are immediately relayed to all affected images and, if "world.instant_update" is true, to all affected displays. The procedure update_images handles this operation.

```
OPERATOR add_to_figure(f:figure_transform;new:figure_transform);

  PRE
    f in world.figure_set;
    new in world.figure_set;
    new != f;
    IF NOT(f.figure in world.primitive_set) THEN
       NOT new in f.figure;
    ENDIF;
```

```
  POST
    IF (f.figure in world.primitive_set) THEN
       LET f1:figure_transform(f.figure,f.trans);
       LET y:VECTOR OF figure_transform | element(y,1) = f1 AND
                                          length(y) = 1;
       f.figure = y;
       LET t:transform(1,0,0,1,0,0);
       f.trans = t;
    ENDIF;
    element(f.figure,length(f.figure)+1) = new;
    update_images(f);
END;


OPERATOR remove_from_figure(f:figure_transform;old:figure_transform);
  PRE
    f in world.figure_set;
    NOT(f.figure in world.primitive_set);
    old in f.figure;
  POST
    shrinkvector(f.figure,old);
    /* NOT old in f.figure */
    update_images(f);
END;


OPERATOR transform_figure(f:figure_transform;t:transform);
  PRE
    f in world.figure_set;
  POST
    f.trans = t;
    update_images(f);
END;
```

Neither the copy nor the destroy_fig_trans operators require any alterations to accomodate the display considerations discussed in this chapter. The new figure_transform created by the copy operator may not initially belong to the "contents" of any image, so has no effect on what the user sees. Destroy_fig_trans exerts its effects on images and displays via calls to remove_from_image and remove_from_figure. These operators have already been modified to include their influence on what is seen by the user.

The operator select_image does not have any effect on the display of images and figures so does not require any modification in this area. However, as selection is made based on what appears on the display screen, it is important that the "screen" reflect the actual "contents" of the display when an image is selected. As can be seen below, another pre-condition has been added to the select_image operator to ensure that the display is, in fact, up-to-date.

```
OPERATOR select_image(id:input_device);
   PRE
     id.enabled = TRUE;
     id.activated = TRUE;
     EXISTS d:display | d in world.display_set
       {
         d.up_to_date;
         id in d.input_group;
         id.where in d.area;
       };
   POST
     EXISTS d:display | d in world.display AND
                        id in d.input_group
       {
         LET b:boolean |
             b = EXISTS i:image | i in d.contents
                 {
                   id.where in element(i.positions,d.display_id);
                 };
         IF b THEN
           EXISTS i:image | i in d.contents AND
                  id.where in element(i.positions,d.display_id)
             {
               LET t:transform |
               t.apply(i.window) = element(i.positions,d.display_id);
               i.input_information.where = t.rapply(id.where);
               i.input_information.code = id.code;
               i.input_information.string = id.string;
               i.input_information.display_id = d.display_id;
               i.chosen = TRUE;
             };
         ENDIF;
       };
END;
```

Chapter 6

Specification of MacPac

The final step in the design of MacPac is the specification. As discussed in Chapter 3, we will use a specification language developed by Mark Green, GUSL, to define the components of MacPac. A GUSL specification consists of a number of modules. Each module is a state machine containing functions that may be used to access and/or change its state. A module is similar to an abstract data type. In our specification of MacPac we will revisit the objects of the MacPac system. These objects, which may be viewed as abstract data types, will each be defined by a module. The state of an object is represented by its attribute values. Each module will includes functions that allow inquiry on these attributes. The functions and operators discussed in previous chapters that change the state of an object will also be incorporated into the specification of the object.

The specification of the MacPac system has been divided into two sections. The first section presents the specification of the major objects of the MacPac system. These are the world, display, image, figure_transform, input_device, and input_event objects. The last section deals with the specification of the graphical primitives line, polyline, and text. Throughout Chapters 4 and 5 we have made use of a

number of other graphical entities, i.e. transform, raster_display, and colour_map. These entities, however, are not specific to MacPac. They are graphical constructs that may be used by any graphics system. Because of this, the specification of these entities is not included in this chapter. Instead, their specification may be seen in Appendix B.

6.1 MacPac Modules

The specification of the world object may be seen in Figure 6.1. In MacPac, only one instance of this module per application is permitted. Each application using MacPac must initialize the world in which it will operate. This includes specification of all display devices to be used and their corresponding input devices. The implementation of the MacPac system must provide a mapping from the devices initialized by the application to the actual physical devices available. The implementation must also provide the routines necessary to convert output from MacPac to a form understandable to each display device and input to MacPac to a form understandable to MacPac.

As can be seen, it is possible to inquire on all world attributes. A new attribute, no_of_display, is introduced in this specification. This attribute must be provided when the world object is created along with the desired display_set. Two new functions, instant_update_on and instant_update_off, have also been added. These functions allow the user to choose the instant_update alternative best suited to his application. There are no restrictions on when these functions may be invoked. However, if instant_update is FALSE and in-

stant_update_on is invoked, all displays must be brought up-to-date. The display function refresh_display, discussed later in this section, is used to handle this requirement.

The world module handles the creation of all images, figures, and graphical primitives which are required by an application. The creation functions for images and figures are similar to the corresponding operators of Chapter 5. The only significant change is the addition of an attribute to the image and figure_transform objects. This new attribute is an integer identification code which is unique for every instance of an object. The two hidden functions, max_image and max_figure, keep track of all identification codes previously assigned. Every time a new image or figure_transform is created, the value of the appropriate function is incremented, and the new value is assigned as the identification code of the new object. The main purpose of this identification code is to ease communication between object instances. For example, if an image instance is altered and we want to update all affected displays with the new image information, the appropriate update function of each display must be invoked. The image identification code is sent as argument to this display update function to indicate which image has been changed. Graphical primitives are created in much the same way as images and figure_transforms. The creation function is called with the desired attribute values for the primitive. A new instance of the primitive object is then created and the primitive is added to the world.primitive_set. Graphical primitives do not require the assignment of an identification code. When one

of these entities is no longer required it may be removed from the world module through use of the appropriate destruction function. Note that when a graphical primitive is destroyed, all elementary figure_transforms whose figure attribute is this primitive are also destroyed.

```
MODULE world;

   PARAMETERS
     hardware : SET OF display;
     max_display : integer;

   FUNCTIONS

     VFUN instant_update -> boolean;
       INITIALLY
         instant_update = TRUE;
     END;

     VFUN display_set -> SET OF display;
       INITIALLY
         display_set = hardware;
     END;

     VFUN image_set -> SET OF image;
       INITIALLY
         image_set = EMPTY;
     END;

     VFUN figure_set -> SET OF figure_transform;
       INITIALLY
         figure_set = EMPTY;
     END;

     VFUN primitive_set -> SET OF primitive;
       INITIALLY
         primitive_set = EMPTY;
     END;

     VFUN no_of_display -> integer;
       INITIALLY
         no_of_display = max_display;
     END;
```

```
VFUN max_image -> integer;
  INITIALLY
  HIDDEN
    max_image = 0;
END;

VFUN max_figure -> integer;
  INITIALLY
  HIDDEN
    max_figure = 0;
END;

OFUN instant_update_on;
  POST
    IF instant_update = FALSE THEN
      FORALL d:display | d in display_set
        {
          d.refresh_display;
        };
    ENDIF;
    instant_update = TRUE;
END;

OFUN instant_update_off;
  POST
    instant_update = FALSE;
END;

OVFUN create_image(s:extent;b,st,rg:integer) -> image;
  PRE
    0 <= b <= rg;
    0 <= st;
  POST
    max_image = max_image + 1;
    LET create_image:image(max_image,s,b,st,rg);
    create_image.init_image;
    create_image.init_picture;
    create_image in image_set;
END;

OFUN destroy_image(i:image);
  PRE
    i in image_set;
  POST
    FORALL d:display | d in display_set AND i in d.contents
      {
        d.erase_image(i);
      };
    NOT i in image_set;
END;
```

```
OVFUN create_fig_trans(e:element;t:transform) ->
                                        figure_transform;
  POST
    IF NOT e in world.primitive_set THEN
      LET e:VECTOR OF figure_transform | length(e) = 0;
    ENDIF;
    max_figure = max_figure + 1;
    LET create_fig_trans:figure_transform(max_figure,e,t);
    create_fig_trans in figure_set;
END;

OFUN destroy_fig_trans(f:figure_transform);
  PRE
    f in figure_set;
  POST
    FORALL f2:figure_transform | f2 in figure_set AND
                                 f in f2.figure
      {
        f2.remove_from_figure(f);
      };
    FORALL i:image | i in image_set AND
                     f in i.contents
      {
        i.remove_from_image(f);
      };
    NOT f in figure_set;
END;

OVFUN create_line(end1,end2:point;col:integer) -> line;
  POST
    LET create_line:line(end1,end2,col);
    create_line in world.primitive_set;
END;

OFUN destroy_line(l:line);
  PRE
    l in world.primitive_set;
  POST
    FORALL f:figure_transform | f.figure = l
      {
        destroy_fig_trans(f);
      };
    NOT l in world.primitive_set;
END;
```

```
    OVFUN create_polyline(endpts:VECTOR OF point;col:integer) ->
                                                          polyline;
      POST
        LET create_polyline:polyline(endpts,col);
        create_polyline in world.primitive_set;
    END;

    OFUN destroy_polyline(pl:polyline);
      PRE
        pl in world.primitive_set;
      POST
        FORALL f:figure_transform | f.figure = pl
          {
            destroy_fig_trans(f);
          };
        NOT pl in world.primitive_set;
    END;

    OVFUN create_text(where:point;space:integer;char_set:font;
                      char_cds:VECTOR OF integer;col:integer) -> text;
      POST
        LET create_text:text(where,space,char_set,char_cds,col);
        create_text in world.primitive_set;
    END;

    OFUN destroy_text(t:text);
      PRE
        t in world.primitive_set;
      POST
        FORALL f:figure_transform | f.figure = t
          {
            destroy_fig_trans(f);
          };
        NOT t in world.primitive_set;
    END;


END MODULE world;
```

Figure 6.1 - World Specification

The next module to be specified is the display. As mentioned earlier, all instances of the display module that are to be available to an application must be defined to the world module when it is

initialized. In defining a display, the display_id and the area, in display coordinates, of the display must be provided. Also, all input devices which will be available to the display must be specified.

The functions of each display control what is seen on the screen of that display. Included in the display functions are the four display alteration operators discussed in Chapter 5; refresh_display, display_image, erase_image, and move_image. As very little manipulation of these operators was required for their incorporation into the display specification, we will not discuss them further. Several of the common functions defined in Chapter 5 may also be used to access or alter the state of a display. They have therefore been included in the display specification. Upper_image has been included with few changes to its original form. Shrinkcontents performs the same function as shrinkvector, but operates specifically on the contents vector of the display. Update_displays takes in an image_id as parameter and, if the display contains an image with this image_id, updates the display screen with respect to this image.

A new function, set_colour_table, is introduced in the display specification. This function may be used to define the colour_table attribute of a display. Another function, outdated, provides a facility whereby a display may be flagged as being not up-to-date with respect to the images it contains.

```
MODULE display;

   PARAMETERS
     d_id : integer;
     d_area : extent;
     input_hardware : SET OF input_device;

   DECLARATIONS
     screen_raster : raster(truncate(d_area.lower_left),
                            truncate(d_area.upper_right));

   DEFINITIONS
     in_range(p) is area.lower_left.x < p.x < area.upper_right.x
                AND area.lower_left.y < p.y < area.upper_right.y;

   FUNCTIONS

     VFUN display_id -> integer;
       INITIALLY
         display_id = d_id;
     END;

     VFUN area -> extent;
       INITIALLY
         area = d_area;
     END;

     VFUN contents -> VECTOR OF image;
       INITIALLY
         contents = EMPTY;
     END;

     VFUN screen -> raster_display;
       INITIALLY
        screen = screen_raster;
     END;

     VFUN up_to_date -> boolean;
       INITIALLY
        up_to_date = TRUE;
     END;

     VFUN input_group -> SET OF input_device;
       INITIALLY
         input_group = input_hardware;
     END;
```

```
VFUN colour_table -> colour_map;
   INITIALLY
     colour_table = ?
END;

OFUN init;
  POST
    FORALL p:point | p in screen
      {
        screen.set_pix(p,0);
      };
END;

OFUN set_colour_table(new_colours:colour_map);
   POST
     colour_table = new_colours;
END;

OFUN outdated;
  POST
    up_to_date = FALSE;
END;

OFUN refresh_display;
  POST
    init;
    FORALL i:image | i in contents
      {
        LET t:transform |
            t.apply(i.window) = i.viewport(d_id);
        FORALL p:point | in_range(p) AND
                         p in i.viewport(d_id)
          {
            IF upper_image(i,p) THEN
              LET c:integer |
                  c = i.picture.pixel(t.rapply(p)) +
                      i.colour_start;
              IF c > colour_table.max THEN
                 c = 0;
              ENDIF;
              screen.set_pix(p,c);
            ENDIF;
          }
      };
    up_to_date = TRUE;
END;
```

```
OFUN update_display(i_id:integer);
  POST
    LET b:boolean |
        b = EXISTS i:image | i in contents
              {
                i.image_id = i_id;
              }:
    IF b THEN
      EXISTS i:image | i in contents AND
                       i.image_id = i_id
        {
          LET t:transform |
              t.apply(i.window) = i.viewport(d_id);
          FORALL p:point | in_range(p) AND
                           p in i.viewport(d_id)
            {
              IF upper_image(i,p) THEN
                LET c:integer |
                    c = i.picture.pixel(t.rapply(p)) +
                        i.colour_start;
                IF c > colour_table.max THEN
                   c = 0;
                ENDIF;
                screen.set_pix(p,c);
              ENDIF;
            };
        };
    ENDIF;
END;

VFUN upper_image(i:image;p:point) -> boolean;
   PRE
     i in contents;
     p in i.viewport(d_id);
   DERIVED
     LET int1:integer | i = element(contents,int1);
     upper_image =
       FORALL i2:image | i2 in contents AND
                         p in i2.viewport(d_id) AND
                         i2 != i
         {
           LET int2:integer | i2 = element(contents,int2);
           int2 < int1;
         };
END;
```

```
OFUN shrinkcontents(i:image);
  PRE
    i in contents;
  POST
    LET index:integer ¦ i = element(contents,index);
    FORALL index2:integer ¦ index <= index2 < length(contents)
      {
        element(contents,index2) = element('contents,index2+1);
      };
    length(contents) = length('contents) -1;
END;

OFUN display_image(i:image;e:extent);
  PRE
    IF i in contents THEN
      e = i.viewport(d_id);
    ENDIF;
    EXISTS p:point ¦ p in e
      {
        in_range(p);
      };
  POST
    IF NOT (element(contents,length(contents)) = i) THEN
      IF i in contents THEN
        shrinkcontents(i);
      ENDIF;
      element(contents,length(contents)+1) = i;
    ENDIF;
    i.set_viewport(d_id,e);
    IF world.instant_update THEN
      LET t:transform ¦ t.apply(i.window) = e;
      FORALL p:point ¦ in_range(p) AND p in e
        {
          LET c:integer ¦
              c = i.picture.pixel(t.rapply(p)) +
                  i.colour_start;
          IF c > colour_table.max THEN
             c = 0;
          ENDIF;
          screen.set_pix(p,c);
        };
    ELSE
      outdated;
    ENDIF;
END;
```

```
    OFUN erase_image(i:image);
      PRE
        i in contents;
      POST
        shrinkcontents(i);
        i.set_viewport(d_id,NULL);
        IF world.instant_update THEN
          refresh_display;
        ELSE
          outdated;
        ENDIF;
    END;

    OFUN move_image(i:image;e:extent);
      PRE
        i in contents;
        EXISTS p:point | p in e
          {
            in_range(p);
          };
      POST
        i.set_viewport(d_id,e);
        IF world.instant_update THEN
          refresh_display;
        ELSE
          outdated;
        ENDIF;
    END;

END MODULE display;
```

Figure 6.2 - Display Specification

The specification for the image object may be seen in Figure 6.3. As mentioned earlier in this section an identifier attribute, image_id, has been added to this module. Functions have been provided to allow inquiry on all image attributes: image_id, contents, positions, window, picture, altered, background, colour_start, colour_range, fill_info_points, fill_info_colours, chosen, and input_information.

The input parameters required for creation of an image are an extent and four integers. These parameters represent the desired values for the image attributes window, image_id, background, colour_start, and colour_range. When a new instance of the image module is created, via the world.create_image function, the image is initialized by the functions init_image and init_picture. Init_image initializes the positions, contents, fill_info_points, and fill_info_colours vectors. The positions vector, which has an entry for each display in the world.display_set, is initially filled with NULL values. The other three vectors are created but, at this point, do not contain any entries. Init_picture initializes the picture attribute of the image, setting each pixel to the background color.

The five image alteration functions discussed in Chapter 5, add_to_image, remove_from_image, pan_image, add_fill_info, and remove_fill_info, are also included in the specification. These functions mirror the operators of the previous chapter so require little comment here. When one of these functions is invoked, it may be necessary to update all displays containing this image to reflect the changes to the image. To accomplish this, the update_display function of each affected display is invoked with the image_id of the image as parameter. As was seen in the display specification, the update_display function will update the display with respect to this image. This is an good example of the usefulness of image_id in providing a means of communicating between modules.

The functions update_image_picture, update_fill, invoke_fill, upper_figure, shrinkvector, and update_images, specified separately in Chapter 5, have been incorporated into the image module to the extent that they operate on image objects. Update_images is the only function that required substantial changes in this incorporation process. The update_images function defined in Chapter 5 takes in a figure_transform and updates all images which contain this figure_transform. The function also updates all images containing a parent (or grandparent, etc) of this input figure_transform via a recursive call to update_images with the parent figure_transform as input. Update_images has been split into two parts in its incorporation into the specification of MacPac. The update_image function of the image module takes a figure_transform identifier as input. If the image contains a figure_transform with this identifier, the image picture and all affected display screens are updated as required. As we will see in the figure_transform specification, this module also contains an update_images function. After a figure_transform has been altered, this function calls the update_image function of each existing image to handle the update of all affected image pictures and display screens. The update_images function of the figure_transform then relays the changes to all parent figure_transforms via a call to the update_images function of each parent.

Additionally, five new functions have been provided for manipulation of images. Two of these functions, set_viewport and viewport, operate on the positions attribute of the image. Set_viewport

may be used to specify a new viewport for a particular display. Viewport may be used to inquire on the viewport which currently exists for a display. The function reset_background_colour allows the user to change the background colour of an image. The new colour specified as input to this function must be within the existing colour range. Both the colour_start and colour_range attributes of an image may be changed using the function reset_colour_submap. If, after the colour_range is changed, the background colour is greater than the colour_range, the background is set to colour 0. An interesting use of this function is to selectively control the visibility of figure_transforms which exist in the image window. We saw in Chapter 5 that if the colour of a figure_transform is greater than the colour_range, the figure_transform is displayed in colour 0. If 0 is the background colour of the image, this figure_transform will not be seen even though it has been added to the image picture. By changing the colour_range the figure can be made to appear. When either of these colour control functions is invoked, the image picture attribute and, if world.instant_update is true, the display screens containing this image are updated to reflect the new colour information. The fifth function, receive_input, simply sets the input_information attribute of the image to the value of the input_event provided as input parameter to this function. Also, the attribute 'chosen' is set to true.

```
MODULE image;

   PARAMETERS
     i_id    : integer;
     i_wndw  : extent;
     i_bkgd  : integer;
     i_cstart : integer;
     i_crange : integer;

   DECLARATIONS
     picture_raster : raster_display(truncate(i_wndw.lower_left),
                                     truncate(i_wndw.upper_right));

   FUNCTIONS

     VFUN image_id -> integer;
       INITIALLY
         image_id = i_id;
     END;

     VFUN contents -> VECTOR OF figure_transform;
       INITIALLY
         contents = ?;
     END;

     VFUN positions -> VECTOR OF extent;
       INITIALLY
         positions = ?;
     END;

     VFUN window -> extent;
       INITIALLY
         window = i_wndw;
     END;

     VFUN picture -> raster_display;
       INITIALLY
         picture = picture_raster;
     END;

     VFUN altered -> boolean;
       INITIALLY
         altered = FALSE;
     END;

     VFUN background -> integer;
       INITIALLY
         background = i_bkgd;
     END;
```

```
VFUN colour_start -> integer;
  INITIALLY
    colour_start = i_cstart;
END;

VFUN colour_range -> integer;
  INITIALLY
    colour_range = i_crange;
END;

VFUN fill_info_points -> VECTOR OF point;
  INITIALLY
    fill_info_points = ?
END;

VFUN fill_info_colours -> VECTOR OF integer;
  INITIALLY
    fill_info_colours = ?
END;

VFUN chosen -> boolean;
  INITIALLY
    chosen = FALSE;
END;

VFUN input_information -> input_event;
  INITIALLY
    input_information = NULL;
END;

VFUN viewport(d_id:integer) -> extent;
  DERIVED
    viewport = element(positions,d_id);
END;

VFUN upper_figure(f:figure_transform;p:point) -> boolean;
  PRE
    f in contents
    f.trans.rapply(p) in f
  DERIVED
    LET int1:integer | f = element(contents,int1);
    upper_figure =
      FORALL f2:figure_transform | f2 in contents AND
                                   f2.trans.rapply(p) in f2 AND
                                   f2 != f
        {
          LET int2:integer | f2 = element(contents,int2);
          int2 < int1;
        };
END;
```

```
OFUN set_viewport(d_id:integer;e:extent);
  POST
    element(positions,d_id) = e;
END;


OFUN shrinkcontents(f:figure_transform);
  PRE
    f in contents;
  POST
    LET int:integer | f = element(contents,index);
    FORALL int2:integer | int <= int2 < length(contents)
      {
        element(contents,int2) = element('contents,int2 + 1);
      };
    length(contents) = length('contents) - 1;
END;


OFUN reset_background_colour(newb:integer);
  PRE
    0 <= newb <= colour_range;
  POST
    FORALL p:point | p in picture
      {
        IF picture.pixel(p) = background THEN
          picture.set_pix(p,newb);
        ENDIF;
      };
    IF world.instant_update THEN
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
    background = newb;
END;
```

```
OFUN reset_colour_submap(newstart,newrange:integer);
  PRE
    newstart >= 0;
    newrange >= 0;
  POST
    colour_start = newstart;
    colour_range = newrange;
    IF background > newrange THEN
       background = 0;
    ENDIF;
    update_image_picture;
    IF world.instant_update THEN
      FORALL d:display ¦ d in world.display_set AND
                          viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display ¦ d in world.display_set AND
                          viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
END;

OFUN init_image;
  POST
    LET p_init:VECTOR OF extent ¦
              length(p_init) = world.no_of_display;
    FORALL int:integer ¦ 0 < int <= length(p_init)
      {
        element(p_init,int) = NULL;
      };
    positions = p_init;
    LET c_init:VECTOR OF figure_transform ¦ length(c_init) = 0;
    contents = c_init;
    LET point_init:VECTOR OF point ¦ length(point_init) = 0;
    fill_info_points = point_init;
    LET colour_init:VECTOR OF integer ¦ length(colour_init) = 0;
    fill_info_colours = colour_init;
END;

OFUN init_picture;
  POST
    FORALL p:point ¦ p in picture
      {
        picture.set_pix(p,background);
      };
END;
```

```
OVFUN add_fill_info(p:point,c:integer) -> integer;
  PRE
    c >= 0;
  POST
    LET next:integer | next = length(fill_info_points) + 1;
    element(fill_info_points,next) = p;
    element(fill_info_colours,next) = c;
    add_fill_info = next;
    IF p in picture THEN
      LET intp:point | intp = truncate(p);
      LET new_colour:integer | new_colour = c;
      IF new_colour > colour_range THEN
         new_colour = 0;
      ENDIF;
      LET old_colour:integer | old_colour = picture.pixel(p);
      invoke_fill(intp,old_colour,new_colour);
      IF world.instant_update THEN
        FORALL d:display | d in world.display_set AND
                           viewport(d.display_id) != NULL
          {
            d.update_display(image_id);
          };
      ELSE
        FORALL d:display | d in world.display_set AND
                           viewport(d.display_id) != NULL
          {
            d.outdated;
          };
      ENDIF;
    ENDIF;
END;

OFUN remove_fill_info(index:integer);
  PRE
    index <= length(fill_info_points);
  POST
    FORALL int:integer | index <= int < length(fill_info_points)
      {
        element(fill_info_points,int) =
                           element('fill_info_points,int+1);
        element(fill_info_colours,int) =
                           element('fill_info_colours,int+1);
      };
    length(fill_info_points) = length('fill_info_points) - 1;
    length(fill_info_colours) = length('fill_info_colours) - 1;
    update_image_picture;
```

```
    IF world.instant_update THEN
      FORALL d:display ¦ d in world.display_set AND
                          viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display ¦ d in world.display_set AND
                          viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
END;


OFUN update_image_picture;

  PRE
    altered = TRUE;

  POST
    init_picture;
    FORALL f:figure_transform ¦ f in contents
      {
        FORALL p:point ¦ p in f AND
                         f.trans.rapply(p) in picture
          {
            IF upper_figure(f,f.trans.apply(p)) THEN

              IF f.colour(p) > colour_range THEN
                picture.set_pix(f.trans.apply(p),0);
              ELSE
                picture.set_pix(f.trans.apply(p),
                                f.colour(p));
              ENDIF;

            ENDIF;
          };
      };
    update_fill;
    altered = FALSE;
END;
```

```
OFUN update_fill;
  POST
    FORALL int:integer ¦ 0 < int <= length(fill_info_points)
      {
        LET p:point ¦ p = truncate(element(fill_info_points,int));
        IF p in picture THEN
          LET new_colour:integer ¦
              new_colour = element(fill_info_colours,int);
          IF new_colour > colour_range THEN
             new_colour = 0;
          ENDIF;
          LET old_colour:integer ¦ old_colour = picture.pixel(p);
          invoke_fill(p,old_colour,new_colour);
        ENDIF;
      };
END;


OFUN invoke_fill(p:point;old,new:integer);
  POST
    IF p in picture THEN
       IF picture.pixel(p) = old THEN
          picture.set_pix(p,new);
          LET p1:point(p.x,p.y+1);
          invoke_fill(p1,old,new);
          LET p2:point(p.x,p.y-1);
          invoke_fill(p2,old,new);
          LET p3:point(p.x+1,p.y);
          invoke_fill(p3,old,new);
          LET p4:point(p.x+1,p.y+1);
          invoke_fill(p4,old,new);
          LET p5:point(p.x+1,p.y-1);
          invoke_fill(p5,old,new);
          LET p6:point(p.x-1,p.y);
          invoke_fill(p6,old,new);
          LET p7:point(p.x-1,p.y+1);
          invoke_fill(p7,old,new);
          LET p8:point(p.x-1,p.y-1);
          invoke_fill(p8,old,new);
       ENDIF;
    ENDIF;
END;
```

```
OFUN add_to_image(f:figure_transform);
  PRE
    f in world.figure_set;
    NOT f in contents;
  POST
    element(contents,length(contents)+1) = f;
    altered = TRUE;
    update_image_picture;
    IF world.instant_update THEN
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
END;


OFUN remove_from_image(f:figure_transform);
  PRE
    f in contents;
  POST
    shrinkcontents(f);
    altered = TRUE;
    update_image_picture;
    IF world.instant_update THEN
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
END;
```

```
OFUN pan_image(w:extent);
  POST
    window = w;
    LET r:raster_display(truncate(window.lower_left),
                         truncate(window.upper_right));
    picture = r;
    altered = TRUE;
    update_image_picture;
    IF world.instant_update THEN
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.update_display(image_id);
        };
    ELSE
      FORALL d:display | d in world.display_set AND
                         viewport(d.display_id) != NULL
        {
          d.outdated;
        };
    ENDIF;
END;


OFUN update_image(f_id:integer);
  POST
    LET b:boolean |
        b = EXISTS f:figure_transform | f in contents
              {
                f.figure_id = f_id;
              };
    IF b THEN
      update_image_picture;
      IF world.instant_update THEN
        FORALL d:display | d in world.display_set AND
                           viewport(d.display_id) != NULL
          {
            d.update_display(image_id);
          };
      ELSE
        FORALL d:display | d in world.display_set AND
                           viewport(d.display_id) != NULL
          {
            d.outdated;
          };
      ENDIF;
    ENDIF;
END;
```

```
    OFUN receive_input(info:input_event);
      PRE
        info.where in window;
      POST
        input_information = info;
        chosen = TRUE;
    END;

END MODULE image;
```

Figure 6.3 - Image Specification

We now turn to the specification of the figure_transform. As can been seen, all three figure_transform attributes, figure_id, figure, and trans, must be provided when defining a figure_transform instance. Thereafter, it is possible to inquire on these attributes via functions of the same name.

Two functions that were used extensively in Chapter 5 are presented in detail here. These functions, 'in' and 'colour', both operate in a recursive fashion. If the figure_transform is an elementary figure_transform, i.e. the figure attribute is a graphical primitive, then the value of each of these functions is simply the value of that function applied to the figure of the figure_transform. If the figure_transform is non-elementary, its figure attribute contains one or more child figure_transforms. In this situation, the value of 'in' or 'colour' function depends on the results of applying that function to each of these children. Before applying one of these functions to a child figure_transform, however, the point provided as input to the function must be transformed into the space of that child.

This is accomplished by applying the inverse of the transform which positions the child within the parent to the input point. The function 'in' is true if a child figure_transform exists that contains the input point once it has been transformed into the coordinate space of that child. Similarly, the function 'colour' returns the colour of the child figure_transform that is upper-most of all those children which contain the input point when it is transformed to the child's space.

The functions upper_figure, shrinkvector, and update_images have been incorporated into this specification. Both upper_figure and shrinkvector, called shrinkfigure in this module, have been included with few changes to their original form. The update_images function found in the figure_transform specification, however, is substantially different from the update_images function of Chapter 5. These differences were pointed out and discussed in the examination of the image object so need not be repeated here.

As for the display and image objects, all operators which manipulate figure_transforms have been included in this specification. These operators, discussed in detail in Chapter 5, required few changes in their incorporation into the figure_transform module.

```
MODULE figure_transform;

   PARAMETERS
     f_id : integer;
     entry : element;
     init_t : transform;
```

```
DEFINITIONS
  boolean elementary IS figure in world.primitive_set;

FUNCTIONS

  VFUN figure_id -> integer;
    INITIALLY
      figure_id = f_id;
  END;

  VFUN trans -> transform;
    INITIALLY
      trans = init_t;
  END;

  VFUN figure -> element;
    INITIALLY
      figure = entry;
  END;

  VFUN in(p:point) -> boolean;
    DERIVED
      IF elementary THEN
        in = figure.in(p);
      ELSE
        in = EXISTS f:figure_transform | f in figure
               {
                 f.trans.rapply(p) in f;
               };
      ENDIF;
  END;

  VFUN colour(p:point) -> integer;
    PRE
      in(p);
    DERIVED
      IF elementary THEN
        colour = figure.colour(p);
      ELSE
        EXISTS f:figure_transform | f in figure AND
                                    f.trans.rapply(p) in f AND
                                    upper_figure(f,p)
          {
            colour = f.colour(f.trans.rapply(p));
          };
      ENDIF;
  END;
```

```
VFUN upper_figure(f:figure_transform;p:point) -> boolean;
  PRE
    NOT elementary;
    f in figure;
    in(p);
    f.trans.rapply(p) in f;
  DERIVED
    LET int1:integer ¦ f = element(figure,int1);
    upper_figure =
      FORALL f2:figure_transform ¦ f2 in figure AND
                                   f2.trans.rapply(p) in f2 AND
                                   f2 != f
        {
          LET int2:integer ¦ f2 = element(figure,int2);
          int2 < int1;
        };
END;

OFUN shrinkfigure(f:figure_transform);
  PRE
    f in figure;
  POST
    LET int1:integer ¦ f = element(figure,int1);
    FORALL int2:integer ¦ int1 <= int2 < length(figure)
      {
        element(figure,int2) = element('figure,int2+1);
      };
    length(figure) = length('figure) - 1;
END;

OFUN update_images;
  POST
    FORALL i:image ¦ i in world.image_set
      {
        i.update_image(f_id);
      };
    FORALL f2:figure_transform ¦ f2 in world.figure_set
      {
        LET b:boolean ¦
            b = EXISTS f:figure_transform ¦ f in f2.figure
                  {
                    f.figure_id = f_id;
                  };
        IF b THEN
           f2.update_images;
        ENDIF;
      };
END;
```

```
OFUN add_to_figure(new:figure_transform);
  PRE
    new in world.figure_set;
    IF NOT elementary THEN
       NOT new in figure;
    ENDIF;
  POST
    IF elementary THEN
      f1 = world.create_fig_trans(figure,trans);
      LET v:VECTOR OF figure_transform ¦ length(v) = 1 AND
                                          element(v,1) = f1;
      figure = v;
      LET t:transform(1,0,0,1,0,0);
      trans = t;
    ENDIF;
    element(figure,length(figure)+1) = new;
    update_images;
END;


OFUN remove_from_figure(old:figure_transform);
  PRE
    NOT elementary;
    old in figure;
  POST
    shrinkfigure(old);
    update_images;
END;


VFUN copy -> figure_transform;
  DERIVED
    IF elementary THEN
      LET y:primitive ¦ y = figure;
      y in world.primitive_set;
      copy = world.create_fig_trans(y,trans);
    ELSE
      LET y:VECTOR OF figure_transform ¦ length(y) = 0;
      copy = world.create_fig_trans(y,trans);
      FORALL f:figure_transform ¦ f in figure
        {
          copy.add_to_figure(f.copy);
        };
    ENDIF;
END;
```

```
    OFUN transform_figure(t:transform);
      POST
        trans = t;
        update_images;
    END;

END MODULE figure_transform;
```

Figure 6.4 - Figure_transform Specification

The final two major MacPac entities requiring specification are the input_device and input_event. The input_device module may be seen in Figure 6.5. Each input_device is associated with a single display. A new attribute, owner_display, that contains the display_id of this associated display, has been added to the definition of this object. The value of this attribute must be provided when an input_device is defined. As discussed earlier, all input_devices that are to be available to a display must be provided when that display instance is created. Also, all displays required by an application must be specified when the world module for that application is created. The effect of this is that all input_device instances that an application will use must be defined when the world module is initialized. Thereafter, no addition or removal of input_devices is permitted.

Although an application may not control the number of input_devices available after the world module is created, input_devices may be selectively enabled and disenabled. The functions enable_for_input and disable_input provide this facility. When an input_device is enabled, it may be activated via the activate_in-

put_device function. For this function to complete successfully, however, the display to which the input_device belongs must be up-to-date. Once activated, the input_device accepts the input information provided by the user. Each implementation of the MacPac system must provide mechanisms capable of handling this transfer of information from the physical input device to the input_device object that represents this physical device. Defaults for missing input information must also be provided by the implementation. After receiving the input information, activate_input_device invokes the function select_image. This function looks for the image which is upper-most on the screen of the associated display at the point received as input. If it finds such an image, the input point is transformed into the image's coordinate space and the input information is relayed, via the receive_input function, to the image. If no image appears on the display screen at this input point, no action is taken. The activation of the input_device is ignored. When the image selection process is complete, activate_input_device sets the activated attribute back to false. This readys the input_device for another activation.

```
MODULE input_device;

   PARAMETERS
     d_id : integer;

   FUNCTIONS

     VFUN owner_display -> integer;
       INITIALLY
         owner_display = d_id;
     END;
```

```
VFUN where -> point;
  INITIALLY
    where = ?;
END;

VFUN code -> integer;
  INITIALLY
    code = ?;
END;

VFUN string -> text;
  INITIALLY
    string = ?;
END;

VFUN enabled -> boolean;
  INITIALLY
    enabled = FALSE;
END;

VFUN activated -> boolean;
  INITIALLY
    activated = FALSE;
END;

OFUN enable_for_input;
  POST
    enabled = TRUE;
END;

OFUN disable_input;
  POST
    enabled = FALSE;
END;

OFUN activate_input_device;
  PRE
    enabled;
    activated = FALSE;
    EXISTS d:display | d.display_id = owner_display
      {
        d.up_to_date;
      };
  POST
    activated = TRUE;
    read(where,code,string);
    select_image;
    activated = FALSE;
END;
```

```
    OFUN select_image;
      PRE
        activated;
      POST
        EXISTS d:display ¦ d.display_id = owner_display
          {
            LET b:boolean ¦
               b = EXISTS i:image ¦ i in d.contents
                    {
                      where in i.viewport(d.display_id);
                    };
            IF b THEN
              EXISTS i:image ¦ i in d.contents AND
                               where in i.viewport(d.display_id) AND
                               d.upper_image(i,where)
                {
                  LET t:transform ¦
                      t.apply(i.window) = i.viewport(d.display_id);
                  LET p:point ¦ p = t.rapply(where);
                  LET input_info:
                      input_event(p,code,string,owner_display);
                  i.receive_input(input_info);
                };
            ENDIF;
          };
    END;

END MODULE input_device;
```

Figure 6.5 - Input_device Specification

The specification of the input_event object is very straightforward. All attributes of this object must be supplied when an input_event instance is defined. Functions are provided to allow inquiry on these attributes after creation. No functions are provided, however, to change the state of an input_event after it is defined. New input_events may be created but an existing input_event instance may not be changed.

```
MODULE input_event;

   PARAMETERS
     p : point;
     cd : integer;
     t : text;
     d_id : integer;

   FUNCTIONS

     VFUN input_display_id -> integer;
       INITIALLY
         input_display_id = d_id;
     END;

     VFUN where -> point;
       INITIALLY
         where = p;
     END;

     VFUN code -> integer;
       INITIALLY
         code = cd;
     END;

     VFUN string -> text;
       INITIALLY
         string = s;
     END;

END MODULE input_event;
```

Figure 6.6 - Input_event Specification

6.2 Graphical Primitives

MacPac supports three basic graphical primitives; line, polyline, and text. The specification of these primitives is presented in this section. In Chapter 3, we discussed the inclusion of a fourth type of graphical primitive, the user-defined primitive. At the end of this section we will look at what is required to include a user-defined primitive in a MacPac application.

There are many similarities in the modules specifying the line and polyline primitives. The parameters required when defining a line are the end points and the colour of the line. To define a polyline, a vector of points and a colour must be provided. Neighboring elements of this vector specify the end points of the set of connecting line segments which make up the polyline primitive. The first and last point of the vector also specify a line segment of the polyline.

The major functions of both modules are 'in' and 'colour'. 'In' takes in a point and returns true if this point is contained in the graphical primitive. For the line module, this amounts to the simple procedure of checking to see if the input point falls on the line. There is nothing new about the mathematics applied in this determination. The polyline module must go a step further and check if the input point falls on any its component line segments. Another function has been added to the polyline module to aid in this operation. This function, in_line_segment, takes in the start and end points of a line segment as well as a point "p". If "p" falls on the line segment specified by the given start and end points, in_line_segment returns true. The function 'in' makes use of this function in its determination of whether or not a given point is contained in the polyline primitive. In_line_segment may also be accessed directly if desired. The start and end points provided as input to the function, however, must be two adjacent points in the vertices attribute of the polyline. The 'colour' functions of both module specifications take a point as input. This point must be contained in the primitive under examination

for successful operation of the function. The value returned by the 'colour' function is that colour specified when the primitive was defined.

```
MODULE line;

   PARAMETERS
     endpoint1,endpoint2 : point;
     col : integer;

   DEFINITIONS
     boolean vertical IS start.x = end.x;
     real x_range IS |(end.x - start.x)|;
     real y_range IS |(end.y - start.y)|;
     boolean in_range_x(p) IS |(p.x - end.x)| <= x_range AND
                              |(p.x - start.x)| <= x_range;
     boolean in_range_y(p) IS |(p.y - end.y)| <= y_range AND
                              |(p.y - start.y)| <= y_range;

   FUNCTIONS

     VFUN start -> point;
       INITIALLY
         start = endpoint1;
     END;

     VFUN end -> point;
       INITIALLY
         end = endpoint2;
     END;

     VFUN slope -> real;
       INITIALLY
         slope = IF NOT vertical THEN
                    (end.y - start.y) / (end.x - start.x);
                 ENDIF;
     END;

     VFUN in(p:point) -> boolean;
       INITIALLY
         in = IF vertical THEN
                (p.x = end.x) AND in_range_y(p);
              ELSE
                ((p.y-end.y) = slope*(p.x-end.x)) AND
                in_range_x(p) AND in_range_y(p);
              ENDIF;
     END;
```

```
    VFUN colour(p:point) -> integer;
      PRE
        in(p);
      INITIALLY
        colour = col;
    END;

END MODULE line;
```

Figure 6.7 - Line Specification

```
MODULE polyline;

   PARAMETERS
     corners : VECTOR OF point;
     col : integer;

   DEFINITIONS

   FUNCTIONS

     VFUN vertices -> VECTOR OF point;
       INITIALLY
         vertices = corners;
     END;

     VFUN in(p:point) -> boolean;
       INITIALLY
         in = EXISTS p1_idx,p2_idx:integer |
                              p1_idx <= length(vertices) AND
                              p2_idx <= length(vertices) AND
                              ( ((p2_idx - p1_idx) = 1)  OR
                                ((p1_idx = 1) AND
                                 (p2_idx = length(vertices))) )
                 {
                   in_line_segment(element(vertices,p1_idx),
                                   element(vertices,p2_idx),p);
                 };
     END;
```

```
    VFUN in_line_segment(start,end,p:point) -> boolean;
      PRE
        start in vertices;
        end in vertices;
        EXISTS idx1,idx2:integer | element(vertices,idx1) = start AND
                                   element(vertices,idx2) = end;
          {
            (| idx1 - idx2 | = 1) OR
            ( (idx1 = 1) AND (idx2 = length(vertices)) ) OR
            ( (idx2 = 1) AND (idx1 = length(vertices)) );
          };
      DERIVED
        LET vertical:boolean | vertical = (start.x = end.x);
        LET x_range:real | x_range = |(end.x - start.x)|;
        LET y_range:real | y_range = |(end.y - start.y)|;
        LET in_range_x:boolean |
            in_range_x = ( |(p.x - end.x)| <= x_range AND
                           |(p.x - start.x)| <= x_range );
        LET in_range_y:boolean |
            in_range_y = ( |(p.y - end.y)| <= y_range AND
                           |(p.y - start.y)| <= y_range );
        IF NOT vertical THEN
           LET slope:real |
               slope = (end.y - start.y) / (end.x - start.x);
        ENDIF;
        in_line_segment = IF vertical THEN
                             (p.x = end.x) AND
                             in_range_y;
                          ELSE
                             ((p.y-end.y) = slope*(p.x-end.x)) AND
                             in_range_x AND
                             in_range_y;
                          ENDIF;
    END;

    VFUN colour(p:point) -> integer;
      PRE
        in(p);
      INITIALLY
        colour = col;
    END;

END MODULE polyline;
```

Figure 6.8 - Polyline Specification

Before going on to present the specification of the text graphical primitive, two new modules need to be defined. These modules, seen in Figure 6.9, represent the entities character and font. The structure of our text specification is built upon these two modules. This breakdown of the text entity is based on previous work done by Mark Green [Green 1982b].

The approach to character representation that we have chosen is to view each character as a small raster. The lower left corner of this raster is always (0,0). The upper right corner of the raster is specified by the input parameters, max_x and max_y. These allow the user to specify the size of the character. A character does not hold any colour information. The character raster contains only the values 0 and 1. Any point with value 1 is considered to be 'in' the character. The user may specify the points that are to make up the character via the char_spot function.

Characters are organized into fonts. The size of the characters to be held in the font, represented by max_x and max_y, and the number of characters in the font must be provided when defining a font instance. Once defined, the font may be initialized by the function init_font. This function initializes the character_set vector in which the characters are stored. At first, each element of this vector is a raster display of the appropriate size whose pixels are all 0. The function set_char may be used to replace one of these initial characters with a new one. The new character and an index representing where in

the character_set vector it is to be placed must be provided to this function. Only characters whose width and height are the same as the max_x and max_y parameters of the font may be added to the font's character_set. The function get_char takes in an index and returns the character which appears in the character_set at that index.

```
MODULE character;

  PARAMETERS
    max_x,max_y : integer;

  DECLARATIONS
    lower_left : point(0,0);
    upper_right : point(max_x,max_y);
    char_raster : raster_display(lower_left,upper_right);

  DEFINITIONS
    boolean in_range(p)  IS  0 <= p.x <= max_x  AND
                             0 <= p.y <= max_y;

  FUNCTIONS

    VFUN height -> integer;
      INITIALLY
        height = max_y;
    END;

    VFUN width -> integer;
      INITIALLY
        width = max_x;
    END;

    VFUN in(p:point) -> boolean;
      INITIALLY
        in = in_range(p) AND (char_pixel(p) = 1);
    END;

    VFUN char_pixel(p:point) -> integer;
      PRE
        in_range(p);
      INITIALLY
        char_pixel = char_raster.pixel(p);
    END;
```

```
    OFUN char_spot(p:point);
      PRE
        in_range(p);
      POST
        char_raster.set_pix(p,1);
    END;

END MODULE character;


MODULE font;

  PARAMETERS
    max_x,max_y : integer;
    char_no : integer;

  FUNCTIONS

    VFUN character_set -> VECTOR OF character;
      INITIALLY
        character_set = ?;
    END;

    VFUN char_width -> integer;
      INITIALLY
        char_width = max_x;
    END;

    VFUN char_height -> integer;
      INITIALLY
        char_height = max_y;
    END;

    VFUN get_char(idx:integer) -> character;
      PRE
        idx <= char_no;
      DERIVED
        get_char = element(character_set,idx);
    END;

    VFUN set_char(c:character;idx:integer);
      PRE
        idx <= char_no;
        c.height = max_y;
        c.width = max_x;
      DERIVED
        element(character_set,idx) = c;
    END;
```

```
    OFUN init_font;
      POST
        LET v:VECTOR OF character ¦ length(v) = char_no;
        FORALL idx:integer ¦ 0 < idx <= char_no
          {
            LET c:character(max_x,max_y);
            element(v,idx) = c;
          };
    END;

END MODULE font;
```

Figure 6.9 - Character and Font Specifications

Having defined character and font, it is now possible to define the text graphical primitive. Several input parameters are required when defining a text instance. The point in user coordinates where the text is to be located must be specified. This point represents the lower_left corner of the first character in the text character string. The horizontal spacing of the characters must also be provided. The two input parameters which specify the actual contents of the text primitive are a vector of integer character codes and a font. Each element of the vector may be used as an index to the character_set of the font. The character returned is the character that will appear in the text character string. The positioning of the integer character codes in the input vector dictates the positioning of the retrieved characters. The final input parameter is the colour of the text primitive.

As for all our graphical primitives, the major functions of the text module are 'in' and 'colour'. 'In' takes in a point and returns true if that point is in the text primitive. To make this deter-

mination, the function must examine each character represented in the string attribute and check if the given point is in that character. As characters are defined in their own coordinate space, we must manipulate the input point before this check can be made. First, the lower left corner of the character must be determined by calculating its distance from the first character in the string. As the lower left of a character's coordinate space is always (0,0), the input point can be moved into the character's space simply by subtracting the lower left corner of the character, as it is positioned in the text character string, from the input point. We then check to see if this corrected point is contained in the character. If so, then the original point is contained in the text primitive. For all points in the text, the function 'colour' returns the colour specified in the input parameters.

```
MODULE text;

   PARAMETERS
     low_left : point;
     h_space : integer;
     char_set : font;
     char_codes : VECTOR OF integer;
     col_init : integer;

   FUNCTIONS

     VFUN lower_left -> point;
       INITIALLY
         lower_left = truncate(low_left);
     END;

     VFUN chosen_font -> font;
       INITIALLY
         chosen_font = char_set;
     END;
```

```
    VFUN string -> VECTOR OF integer;
      INITIALLY
        string = char_codes;
    END;

    VFUN in(p:point) -> boolean;
      DERIVED
        in = EXISTS idx:integer | 0 < idx <= length(string)
              {
                LET char_source_pt:point |
                    char_source_pt.y = lower_left.y  AND
                    char_source_pt.x = (lower_left.x +
                          ((idx-1) * h_space) +
                          ((idx-1) * chosen_font.char_width));
                LET corrected_pt:point |
                    corrected_pt = (p - char_source_pt);
                LET char_code:integer |
                    char_code = element(string,idx);
                corrected_pt in chosen_font.get_char(char_code);
              };
    END;

    VFUN colour(p:point) -> integer;
      PRE
        in(p);
      INITIALLY
        colour = col_init;
    END;

END MODULE text;
```

Figure 6.10 - Text Specification

Any implementation of the MacPac system should provide a number of predefined fonts. A standardized approach to the ordering of characters within each font character_set should also be taken. In other words, the a's should fall at the same vector index within different font character_sets, and similarly for the b's, c's, etc. This type of standardization facilitates provision of a character decoding scheme. For instance, a table can be set up that assigns the number 1 to the

character a, the number 2 to b, and so on. Then, when creating text, if one enters a string of character codes 'abc', this will be decoded as the vector (1,2,3). This is very valuable as it is much easier for a user to specify text by providing a string of characters as opposed to a vector of integer character codes. Although most applications do not make use of special or unusual character_sets, through the functions of the character and font modules, MacPac provides the user with the facility to create character_sets of his own design.

It should be noted that none of the modules that define the graphical primitives of MacPac contain functions capable of modifying the state of the module. Once a graphical primitive is created it may not be altered. For instance, neither the end points of a line nor the characters contained in a text primitive may be changed. If a line or text is required with new attributes, a new module instance must be created. As was seen in the previous section, when a graphical primitive is no longer required it may be destroyed via the appropriate world destroy function.

We now turn to the fourth type of graphical primitive, the user-defined primitive. There are very few restrictions on the inclusion of user-defined primitives in a MacPac application. When defining a new primitive type, the user must provide a specification for this primitive similar to what we have provided for line, polyline, and text. The input parameters required for declaration of a primitive instance must be specified. Also, the functions 'in' and 'colour' must be

provided. It is the function 'in' which places the greatest restriction on user-defined primitives. Obviously, a function capable of determining whether or not a point falls within the graphical primitive must exist. In effect, a scan conversion function for the primitive must be provided. An alternative to this is to require that a complete bit map of the primitive be provided when a primitive instance is defined. The function 'colour' is relatively straightforward to supply. Either the user may require that the colour of the primitive be provided as a separate input parameter, or that the bit map for the primitive contain colour information. The user must also extend the world module to provide creation and destruction operators for the user-defined primitive instances. The world.primitive_set must be kept up-to-date by these operators.

This concludes the specification of the MacPac system. Obviously, from any design, several implementation pathways are possible. The one chosen will invariably depend on the programmer and the types of applications expected to make use of the graphics system. The next chapter will touch on some important considerations in the implementation of MacPac in the process of evaluating the design of this system.

Chapter 7

Conclusions

The primary goal of this work was to design a graphics package based on an existing design philosophy and using a specific design language; both created by Mark Green. Within this framework, the graphics package was to address the hardware and software ideas of the '80's, incorporating where appropriate the valuable and tested ideas of existing systems. MacPac is the result of these efforts. In this chapter we examine the extent to which MacPac meets its design objectives.

A major goal in the creation of MacPac was to address the raster display hardware predominant today. The fact that MacPac fulfills this goal is readily seen in the display representation adopted for the system; the "raster_display" construct. All information required to produce a picture on a display is held in a form compatible with that used for the refresh buffer of the majority of raster display devices. This representation allows MacPac to be extremely device independent. As nearly all raster display devices allow direct loading of the refresh memory, it is a simple matter to create a device driver to add a new device to a system. Also, few application changes would be required to accomodate this new device. MacPac's display representation

is particularly compatible with the single address-space architecture seen in many raster display devices today [Acquah 1982]. Under this architecture, the raster refresh buffer is contained within the memory space of the host computer. In this situation, the "screen" attribute of each MacPac display would actually be the refresh buffer for that display.

As always, a trade-off is found in this choice of display representation. To achieve this level of portability, a sacrifice is made in the ability of an application program to make use of special hardware features. If this is of grave importance to the applications at a particular site, however, the structure of MacPac is sufficiently flexible that it may be easily modified to fulfill this requirement. A command file may be incorporated into the image object and the routines which maintain the image "picture" and display "screen" are easily modified to handle this file. In this situation, user-defined graphical primitives would require specification of the hardware commands required to generate the primitive, as well as the 'in' and 'colour' functions discussed in Chapter 6.

Another area in which the structure of MacPac addresses current hardware is in its ready adaptation to a multi-processor environment. Each image of MacPac may be treated as an independent process. The image may receive and process input information without needing to be aware of what other images are responsible for or involved in. Although the MacPac system currently allows only a single image to be processing

input information at any one time, no structural change is required to allow for the possibility of concurrent image processing.

A second goal of MacPac was to address current ideas in graphics software. This objective has been realized in several areas. Of major importance is the inclusion of the mechanisms whereby graphical resources may be managed. No MacPac entity may be manipulated unless the manipulator has access to the descriptor for that entity. It is not possible to add, remove, or move an image on a display without access to that display. Similarly, one must possess the descriptor for an image in order to manipulate the attributes of that image. The same holds true for figure_transforms. The hierarchical structure of MacPac provides a framework for resource management. Once an image has been allocated a particular area of a display screen, any alterations to the image will be seen only in this allocated area. The ability to control resource management is seen in other aspects of MacPac as well. For example, an image may be allocated a colour "submap". This submap specifies a portion of the colour_map of each display that contains the image. The effect of this is that the only colours seen within the image's viewport on a display will be those found within the allocated portion of that display's colour_map. Another example may be seen in MacPac's input handling facility. Input information is sent directly to the indicated image (as determined by the point associated with the input), thus ensuring that an image does not have access to irrelevant input information.

Another area in which MacPac addresses current software ideas is in its ability to interface with graphical tools, specifically graphical databases. Graphical information in MacPac is held in a hierarchical structure. Displays contain images, images contain figure_transforms, and figure_transforms contain other figure_transforms and/or graphical primitives. This storage structure is very compatible with a large number of databases. The hierarchical structure also eases the extraction of graphical information from a database when it is required for display purposes. Indeed, an implementor may find that the most viable storage vehicle for the large amount of graphical information maintained by the MacPac system will be a graphical database.

In Chapter 2 we discussed several problems inherent in the methodology of mainstream packages. Major among these problems were considerations of portability and device independence, resource management, input handling and viewing. The manner in which MacPac addresses the first two problems has already been discussed. Here we will take a brief look at MacPac's solutions to the problems of viewing and input handling. The viewing difficulties encountered in mainstream packages are largely due to the idea of a single "open" segment to which all output primitives are added. This "open" segment is accompanied by a single "active transformation". MacPac's images and figure_transforms are always "open". Also, there is no such thing as an "active transformation" in the MacPac system. Graphical information is stored in its defined form along with related transformations. When

it is necessary to (re)create a picture of this information for display purposes, the graphical data and transformations are available to do so. Chapter 2 pointed out problems with the concept of virtual input device, and the resulting segregation of input devices into classes. This problem is not seen in the MacPac system as all input is mapped into a single input class. The problem of separation of input and output, however, has not been addressed by the MacPac system. Possible enhancements in this area include the addition of of echo and prompt attributes to the input_device object. In all of these problem areas, the design of MacPac takes into consideration, and incorporates valuable aspects of, the solutions which existing systems have adopted.

The design of any system intended for widespread use is fraught with difficulties. What is good for one user may create many problems for another. Hardware that is standard at one site may not exist at another. As a result, many compromises and trade-offs must be made in system design. A number of important trade-offs had to be made in the design of MacPac. Portability and device-independence was achieved at the cost of readily available access to hardware facilities. The ability to store the graphical entities of the application relieves the program of the chore of re-defining these entities whenever a new view or picture update is required. This aspect of the system, however, limits the implementation of MacPac to sites that can fulfill its extensive storage requirements. Less significant choices had to be made at every stage in the design of MacPac. For example, when using MacPac, two points and a colour must be provided for declaration of a line. The

ability to specify the colour of each line as it is created will be useful to some users, undesired overhead for others. In most cases, someone will disagree with the chosen solution to a problem. In the design of MacPac we have attempted to chose solutions and pathways which will satisfy the requirements of the majority of users. At the same time, the design is intended to provide some flexibility in implementation possibilities. An implementor may tailor MacPac to respond to the available hardware and intended applications at a particular site.

Finally, we need to consider the effectiveness of the design languages, UML and GUSL, in the development of MacPac. As a descriptive tool, these design languages proved to be very useful. Through the constructs of UML and GUSL it was possible to transform the design ideas of MacPac into a consistent, clear description of the system. The structure of the design description also greatly facilitated error and completeness checking. A problem was encountered, however, in the use of these design languages for the development of a graphics system. The languages were originally created for the design of graphical user interfaces. They were therefore exceptionally useful in describing the interface between MacPac and an application program. When dealing with aspects of the system not directly related to this interface, however, the languages tended to be restrictive. As a result, implementation considerations that may have been incorporated into the design of the system had a different design tool been used, were considered to be external to the design of MacPac.

Ideally, the next step in the development of MacPac would be its implementation. Only in this way is it possible to truly test the usefulness of MacPac for the graphics applications of the '80's. It should not be expected that MacPac will prove to be ideal for all uses and applications. No doubt there will be numerous problems. However, it is through development efforts such as MacPac, and the learning that accompanies the effort, that we will eventually arrive at a viable graphics standard for the hardware and software of the future.

Appendix A

UML and GUSL

1 Language Structure - Constructs

1.1 UML Constructs

The important constructs of UML are the OBJECT, OPERATOR, and INVARIANT. The structure of these constructs is illustrated below.

OBJECT - The attributes of an object may be referenced as 'object_name.attribute_name'. Each attribute must belong to one of the theories (pre- or specifier-defined) of the base language.

```
OBJECT object_name;
   ATTRIBUTE attribute_name : type;
   ATTRIBUTE attribute_name : type;
                .
                .
                .
   ATTRIBUTE attribute_name : type;
END;
```

OPERATOR - an operator may have a number of input parameters. These are specified within brackets in the operator header. Also, the operator may return a value. The type of this value is specified after the arrow in the header. Neither parameters nor a result type are required, however. All types in the header must be one of the theories of the base language. The assertions in the body of the operator are written in the base language. These assertions may be made up of values, base language theories, and/or special forms.

```
OPERATOR operator_name(parameter_name:type) -> type;
   PRE
     assertion;
         .
         .
     assertion;
   POST
     assertion;
     assertion;
         .
         .
     assertion;
END;
```

INVARIANT - The assertion of an invariant is written in the base language. It may contain values, theories, and/or special forms. The value of this assertion must always be true.

```
INVARIANT
   assertion;
```

1.2 GUSL Constructs

The major construct of GUSL is the MODULE. The structure of a MODULE is shown below. This is followed by information pertinent to the components of this construct.

```
MODULE module_name;

   PARAMETERS
     parameter_name : type;
     parameter_name : type;
           .
           .
     parameter_name : type;

   DECLARATIONS
     variable_name : type;
     variable_name : type;
           .
           .
     variable_name : type;
```

```
DEFINITIONS
  type definition_name(parameter_names) IS expression;
  type definition_name(parameter_names) IS expression;
         .
         .
  type definition_name(parameter_names) IS expression;

FUNCTIONS

  VFUN function_name(parameter_name:type) -> type;
    PRE
      assertion;
      assertion;
         .
         .
      assertion;
    INITIALLY
      function_name = value;
  END;

  VFUN function_name(parameter_name:type) -> type;
    PRE
      assertion;
      assertion;
         .
         .
      assertion;
    INITIALLY
    HIDDEN
      function_name = value;
  END;

  VFUN function_name(parameter_name:type) -> type;
    PRE
      assertion;
      assertion;
         .
         .
      assertion;
    DERIVED
      assertion;
      assertion;
         .
         .
      assertion;
  END;
```

```
    OFUN function_name(parameter_name:type);
      PRE
        assertion;
        assertion;
           .
           .
        assertion;
      POST
        assertion;
        assertion;
           .
           .
        assertion;
    END;

    OVFUN function_name(parameter_name:type) -> type;
      PRE
        assertion;
        assertion;
           .
           .
        assertion;
      POST
        assertion;
        assertion;
           .
           .
        assertion;
    END;

END MODULE module_name;
```

All types used in a GUSL module must belong to one of the theories of the base language. In other words, each type used in a GUSL specification must be either a pre-defined theory of the base language or must be defined by a GUSL module.

The functions of a module may be referenced in the following way : 'module_name.function_name(parameters)'. The pre-conditions section of each function is optional. In the DERIVED section of a VFUN or the POST section of an OVFUN, at least one of the assertions must as-

sign a value to function_name. VFUNs with the keyword INITIALLY are used to access the state of the module. The value of one of these VFUNs may only be changed by an OFUN of the same module. If the keyword HIDDEN is used, the VFUN may only be accessed from within the module. All assertions of a module function are written in the base language. These assertions may be made up of values, base language theories, and/or special forms. The macros defined in the DEFINITIONS section of the module may also be used in these assertions.

1.3 The Base Language - Special Forms

Following are the special forms used in the base language. Each special form has the value true or false depending on the value of the assertion(s) contained within the special form.

LET - The LET special form attempts to create a new instance of type 'type_name' which satisfies the given assertions. If this is possible, the value of the special form is true.

```
LET variable_name : type_name ¦ assertion;
```

FORALL - The FORALL special form is true if the assertions in body of the special form are true for all entities of type 'type_name' that satisfy 'function(variable_name)'. The variable, 'variable_name', is local to the special form. 'Function(variable_name)' is simply an assertion involving 'variable_name'.

```
FORALL variable_name : type_name ¦ function(variable_name)
  {
    assertions;
  };
```

EXISTS - The EXISTS special form is true if the assertions in its body are true for at least one value of 'variable_name' that satisfies 'function(variable_name)'. The variable, 'variable_name', is local to the special

form. 'Function(variable_name)' is simply an assertion involving 'variable_name'.

```
EXISTS variable_name : type_name | function(variable_name)
  {
    assertions;
  };
```

IF - The IF special form is used to choose between two sets of assertions. The choice is based on the value of another assertion, that which falls after the keyword IF. If this assertion is true the first set of assertions is considered. If the value of this assertion is false, the set of assertions after the keyword ELSE is chosen for consideration. The IF special form is true if all assertions in the chosen set are true.

```
IF assertion THEN
  assertions;
ELSE
  assertions;
ENDIF;
```

SELECT - The SELECT special form is used to choose between a number of sets of assertions. The assertion after each CASE keyword is examined. If the assertion is true then the set of assertions associated with that CASE assertion are chosen. The value of the SELECT special form is true if all assertions in the chosen set are true.

```
SELECT
  CASE assertion  { assertions; };
  CASE assertion  { assertions; };
          .
          .
          .
  CASE assertion  { assertions; };
END SELECT;
```

2 Pre-defined Theory Operators

INTEGER -

arithmetic operators : + , - , * , /

comparison operators : < , <= , = , != , >= , >

```
REAL -
    arithmetic operators :   + , - , * , /
    comparison operators :   < , <= , = , != , >= , >

BOOLEAN -
    AND
    OR
    NOT

SET -
    in : takes in an entity of the type contained in the set and
         returns true if that entity is in the set.
    union : takes in a set and returns the union of this set and
         the set whose union function is being invoked.
    intersect : takes in a set and returns the intersection of
         this set and the set whose intersect function is being
         invoked.

VECTOR -
    in : takes in an entity of the type contained in the vector
         and returns true if that entity is in the vector.
    length : returns the length of the vector.
    element : takes in an integer and uses this integer as an
         index to the vector.  The element of the vector that
         is located at this index is returned.

POINT -
    arithmetic operators :   + , -

EXTENT -
    in : takes in a point and returns true if that point falls
         within the area defined by the extent.
```

3 Grammars

The grammars that follow are taken directly from the appendices of Mark Green's technical report "A Specification Language and Design Notation for Graphical User Interfaces" [Green 1981]. The base language grammar is presented first, followed by the grammars of UML and GUSL. Both UML and GUSL build upon the constructs of the base language grammar, adding further contructs of their own to create a more specialized language.

A number of notational conventions are employed in these grammars. The use of square brackets indicates that anything within the brackets is optional. Curly brackets, or braces, indicate that the what is contained within the brackets may be repeated zero or more times. All terminals are enclosed in double quotes.

3.1 Base Language Grammar

```
base_type        ::= "integer"
                   ¦ "real"
                   ¦ "string"
                   ¦ "point"
                   ¦ "extent"
                   ¦ "boolean"
                   ¦ "set_type"
                   ¦ "vector_type"
                   ¦ "enumeration_type";

set_type         ::= "SET OF" type;

vector_type      ::= "VECTOR OF" type;

enumeration_type ::= "(" literal {"," literal} ")";

literal          ::= identifier;

assertion        ::= expression;

expression       ::= value
                   ¦ special_form
                   ¦ "(" expression ")"
                   ¦ expression "=" expression;

special_form     ::= let_form
                   ¦ forall_form
                   ¦ exists_form
                   ¦ if_form
                   ¦ select_form;

let_form         ::= "LET" variable_name ":" type "¦" assertion;

forall_form      ::= "FORALL" variable_name ":" type ["¦" assertion]
                       "{" {assertion ";"} "}";
```

```
exists_form      ::=  "EXISTS" variable_name ":" type ["|" assertion]
                        "{" {assertion ";"} "}";

if_form          ::=  "IF" assertion "THEN" {assertion ";"}
                        ["ELSE" {assertion ";"}] "ENDIF";

select_form      ::=  "SELECT" {case_item} "END SELECT";

case_item        ::=  "CASE" assertion "{" {assertion ";"} "}";

variable_name    ::=  identifier;
```

3.2 UML Grammar

```
task_model             ::=  task_model_header
                              [type_declarations]
                              {object_definition}
                              {operator_definition}
                              {invariant}
                              task_model_trailer;

task_model_header      ::=  "TASK MODEL" identifier ";";

task_model_trailer     ::=  "END TASK MODEL" identifier ";";

control_model          ::=  control_model_header
                              [type_declarations]
                              {object_definition}
                              {operator_definition}
                              {invariant}
                              control_model_trailer;

control_model_header   ::=  "CONTROL MODEL" identifier ";";

control_model_trailer  ::=  "END CONTROL MODEL" identifier ";";

type_declarations      ::=  "TYPES" {type_dec};

type_dec               ::=  identifier "=" type ";";

object_definition      ::=  "OBJECT" object_name
                              {attribute_definition}
                              "END;";

object_name            ::=  identifier;

attibute_definition    ::=  "ATTRIBUTE" attribute_name ":" type ";";
```

```
attribute_name          ::=  identifier;

operator_definition     ::=  operator_header
                               [pre_part]
                               post_part
                               "END;";

operator_header         ::=  "OPERATOR" operator_name
                               [parameter_list]
                               ["->" type] ";";

operator_name           ::=  identifier;

parameter_list          ::=  "(" parameter_declaration
                               {";" parameter_declaration} ")";

parameter_declaration   ::=  parameter_name ":" type;

parameter_name          ::=  identifier;

pre_part                ::=  "PRE" {assertion ";"};

post_part               ::=  "POST" {assertion ";"};

invariant               ::=  "INVARIANT" assertion ";";

type                    ::=  base_type
                           ¦  object_name
                           ¦  theory_name;

value                   ::=  object_name
                           ¦  value "." attribute_name
                           ¦  parameter_name
                           ¦  expression theory_operator expression
                           ¦  expression "." theory_operator
                           ¦  expression "." theory_operator
                                "(" expression_list ")";

expression_list         ::=  expression {"," expression};
```

3.3 GUSL Grammar

```
module                  ::= module_header
                              [parameter_declarations]
                              [variable_declarations]
                              [macro_definitions]
                              function_definitions
                              module_trailer;

module_header           ::= "MODULE" module_name;

module_trailer          ::= "END MODULE" module_name;

module_name             ::= identifier;

parameter_declarations  ::= "PARAMETERS" {parameter_dec};

parameter_dec           ::= parameter_name ":" parameter_type ";";

parameter_name          ::= identifier;

variable_declarations   ::= "DECLARATIONS" {variable_dec};

variable_dec            ::= variable_name ":" type ";";

variable_name           ::= identifier;

macro_definitions       ::= "DEFINITIONS" {macro_def};

macro_def               ::= type macro_header "IS" expression;

macro_header            ::= macro_name [macro_parameter_list];

macro_name              ::= identifier;

macro_parameter_list    ::= "(" identifier {"," identifier} ")";

function_definitions    ::= "FUNCTIONS" {function_def};

function_def            ::= vfun_def
                          | ofun_def
                          | ovfun_def;

vfun_def                ::= vfun_header
                              [pre_part]
                              vfun_body
                              "END;";
```

```
vfun_header          ::=  "VFUN" function_name
                            [parameter_list] "->" type ";";

function_name        ::=  identifier;

parameter_list       ::=  "(" parm_dec {";" parm_dec} ")";

parm_dec             ::=  parameter_name ":" type;

pre_part             ::=  "PRE" {assertion ";"};

vfun_body            ::=  primitive_body
                        |  derived_body;

primitive_body       ::=  "INITIALLY" ["HIDDEN"]
                            function_name "=" expression ";";

derived_body         ::=  "DERIVED" {assertion ";"};

ofun_def             ::=  ofun_header
                            [pre_part]
                            post_part
                            "END;";

ofun_header          ::=  "OFUN" function_name
                            [parameter_list] ";";

post_part            ::=  "POST" {assertion ";"};

ovfun_def            ::=  ovfun_header
                            [pre_part]
                            post_part
                            "END;";

ovfun_header         ::=  "OVFUN" function_name
                            [parameter_list] "->" type ";";

parameter_type       ::=  base_type
                        |  module_name;

type                 ::=  base_type
                        |  module_name [parameter_values];

parameter_values     ::=  "(" expression  {"," expression} ")";
```

```
value                ::=  function_name
                        ¦  function_name "(" expression_list ")"
                        ¦  parameter_name
                        ¦  variable_name
                        ¦  expression function_name expression
                        ¦  expression "." function_name
                        ¦  expression "." function_name
                             "(" expression_list ")";

expression_list      ::=  expression {"," expression};
```

Appendix B

Basic Graphical Entities

The specifications of the raster_display and transform entities are derived from those presented in [Green 1982b].

1 Raster display Specification

```
MODULE raster_display;

  PARAMETERS
    lower_left : point;
    upper_right : point;

  DEFINITIONS
    boolean in_x_range(p) IS lower_left.x <= p.x <= upper_right.x;
    boolean in_y_range(p) IS lower_left.y <= p.y <= upper_right.y;

  FUNCTIONS

    VFUN frame_store(x,y:integer) -> integer;
      INITIALLY
      HIDDEN
        frame_store = 0;
    END;

    VFUN in(p:point) -> boolean;
      DERIVED
        in = in_x_range(p) AND in_y_range(p);
    END;

    VFUN pixel(p:point) -> integer;
      PRE
        in_x_range(p);
        in_y_range(p);
      DERIVED
        pixel = frame_store(truncate(p.x),truncate(p.y));
    END;
```

```
    OFUN set_pix(p:point;int:integer);
      PRE
        in_x_range(p);
        in_y_range(p);
      POST
        frame_store(truncate(p.x),truncate(p.y)) = int;
    END;

END MODULE raster_display;
```

2 Transform Specification

In the transformation specification that follows, the transformation is represented by six real numbers. These are the components of the transformation matrix :

```
| rs11  rs12  0 |
| rs21  rs22  0 |
| tx    ty    1 |
```

When the transformation is applied to a point (x,y), the transformed point (x',y') is found through the following matrix equation :

```
                                  | rs11  rs12  0 |
| x' y' 1 |  =  | x  y  1 |  *   | rs21  rs22  0 |
                                  | tx    ty    1 |
```

```
MODULE transform;

   PARAMETERS
     rs11_init,rs12_init,rs21_init,rs22_init : real;
     tx_init,ty_init : real;
```

```
FUNCTIONS

  VFUN rs11 -> real;
    INITIALLY
      rs11 = rs11_init;
  END;

  VFUN rs12 -> real;
    INITIALLY
      rs12 = rs12_init;
  END;

  VFUN rs21 -> real;
    INITIALLY
      rs21 = rs21_init;
  END;

  VFUN rs22 -> real;
    INITIALLY
      rs22 = rs22_init;
  END;

  VFUN tx -> real;
    INITIALLY
      tx = tx_init;
  END;

  VFUN ty -> real;
    INITIALLY
      ty = ty_init;
  END;

  OFUN translate(x,y:real);
    POST
      tx = 'tx + x;
      ty = 'ty + y;
  END;

  OFUN scale(x,y:real);
    PRE
      x,y >= 0;
    POST
      rs11 = 'rs11 * x;
      rs12 = 'rs12 * y;
      rs21 = 'rs21 * x;
      rs22 = 'rs22 * y;
      tx = 'tx * x;
      ty = 'ty * y;
  END;
```

```
OFUN rotate(a:integer);
  PRE
    0 <= a < 360;
  POST
    rs11 = ('rs11 * cos(a)) - (rs12 * sin(a));
    rs12 = ('rs11 * sin(a)) + ('rs12 * cos(a));
    rs21 = ('rs21 * cos(a)) - (rs22 * sin(a));
    rs22 = ('rs21 * sin(a)) + ('rs22 * cos(a));
    tx = ('tx * cos(a)) - (ty * sin(a));
    ty = ('tx * sin(a)) + ('ty * cos(a));
END;


OFUN reset_transform;
  POST
    rs11 = rs11_init;
    rs12 = rs12_init;
    rs21 = rs21_init;
    rs22 = rs22_init;
    tx = tx_init;
    ty = ty_init;
END;


OVFUN apply(p:point) -> point
  POST
    LET p1:point |
        p1.x = ((p.x * rs11) + (p.y * rs21) + tx) AND
        p1.y = ((p.x * rs12) + (p.y * rs22) + ty);
    apply = p1;
END;


OVFUN apply(e:extent) -> extent;
  PRE
    /* no rotation */
    rs12 = 0;
    rs21 = 0;
    rs11 >= 0;
    rs22 >= 0;
  POST
    LET p1:point | p1 = apply(e.lower_left);
    LET p2:point | p2 = apply(e.upper_right);
    LET new:extent(p1,p2);
    apply = new;
END;
```

```
    VFUN apply(t:transform) -> transform;
      DERIVED
        LET new_rs11:real |
            new_rs11 = ((rs11 * t.rs11) + (rs12 * t.rs21));
        LET new_rs12:real |
            new_rs12 = ((rs11 * t.rs12) + (rs12 * t.rs22));
        LET new_rs21:real |
            new_rs21 = ((rs21 * t.rs11) + (rs22 * t.rs21));
        LET new_rs22:real |
            new_rs22 = ((rs21 * t.rs12) + (rs22 * t.rs22));
        LET new_tx:real |
            new_tx = ((tx * t.rs11) + (ty * t.rs21) + t.tx);
        LET new_ty:real |
            new_ty = ((tx * t.rs12) + (ty * t.rs22) + t.ty);
        LET new_transform:transform(new_rs11,new_rs12,new_rs21,
                                    new_rs22,new_tx,new_ty);
        apply = new_transform;
    END;

END MODULE transform;
```

3 Colour map Specification

```
MODULE colour_map;

   PARAMETERS
     size : integer;
     colour_init : VECTOR OF integer;

   FUNCTIONS

     VFUN max -> integer;
       INITIALLY
         max = size;
     END;

     VFUN colours -> VECTOR OF integer;
       INITIALLY
         colours = colour_init;
     END;
```

```
    VFUN retrieve_colour(idx:integer) -> VECTOR OF integer;
      PRE
        1 <= idx <= max;
      DERIVED
        LET red_idx:integer ¦ red_idx = ((idx - 1) * 3) + 1;
        LET green_idx:integer ¦ green_idx = red_idx + 1;
        LET blue_idx:integer ¦ blue_idx = green_idx + 1;
        LET v:VECTOR OF integer ¦ length(v) = 3;
        element(v,1) = element(colours,red_idx);
        element(v,2) = element(colours,green_idx);
        element(v,3) = element(colours,blue_idx);
        retrieve_colour = v;
    END;

    VFUN set_colour(idx,red,green,blue:integer);
      PRE
        1 <= idx <= max;
        0 <= red,green,blue <= 255;
      DERIVED
        LET red_idx:integer ¦ red_idx = ((idx - 1) * 3) + 1;
        LET green_idx:integer ¦ green_idx = red_idx + 1;
        LET blue_idx:integer ¦ blue_idx = green_idx + 1;
        element(colours,red_idx) = red;
        element(colours,green_idx) = green;
        element(colours,blue_idx) = blue;
    END;

END MODULE colour_map;
```

Appendix C

Compendium of Operators

1 Display Manipulation Operators

refresh_display - completely recreates the "screen" of a specified display from the information held in the "contents" of that display. First, the "screen" is cleared to colour 0. Each image in the display "contents" is then considered and the "screen" pixels affected by the image are updated using the image's colour information.

display_image - takes a display, an image, and a viewport (extent) as input. If the image is not currently in the "contents" of the display, it is added to the end of the "contents" vector, thereby becoming the image with highest priority in the "contents" of the display. The input viewport, which specifies where on the display "screen" the image is to be seen, is assigned to the image. If the image is currently in the "contents" of the display, it is moved from its current position in the "contents" vector to the end of the vector. The input viewport has no function here as the image remains in the same viewport it currently occupies. If "instant_update" is true, the

display "screen" is updated to reflect this new information. The end result is that the input image will be seen above all others in the portion of the display "screen" specified by the image's viewport.

erase_image - removes a specified image from the "contents" of a given display. If "instant_update" is true, the image is also removed (erased) from the "screen" of the display. Any images which were hidden by this erased image will appear.

move_image - used to change the viewport in which a specified image is seen on a given display. The image, which must currently appear on the "screen" of the specified display, is assigned the viewport provided as input to the operator. If "instant_update" is true, the "screen" of the display is updated to show the image in its new viewport. The priority of the image in the "contents" of the display is not changed.

2 Image Manipulation Operators

create_image - creates a new image instance. The "window", "background", "colour_start", and "colour_range" attributes of the image must be provided as input to the operator. Initially, the image does not contain any figure_transforms and is not seen on any display. The "picture" of the image is defined from the "window" attribute and is initialised to the given "background" colour. The new image instance is added to "world.image_set".

destroy_image - destroys a specified image by removing it from all displays of which it is a member of the "contents" and then removing it from the "world.image_set". This operator does not destroy the figure_transforms of which the image is composed. It simply destroys their relationship as elements of a vector.

add_to_image - adds a specified figure_transform to the "contents" of a given image. The "picture" of the image is updated to reflect this addition. If "instant_update" is true, all displays containing the image are also updated.

remove_from_image - removes a specified figure_transform from the "contents" of a given image. The "picture" of the image is updated to reflect this change to the image's "contents". If "instant_update" is true, all displays containing this image are also updated.

pan_image - changes the "window" attribute of a specified image. The new value for "window" must be provided as input to the operator. The "picture" of the image is completely updated to reflect this new view onto the "contents" of the image. If "instant_update" is true, all displays containing this image are also updated to reflect the new image "picture".

add_fill_info - adds new fill information to a given image. The point and colour which specify the fill must be provided as input to the operator. The point is added to the "fill_info_points"

vector and the colour is added to the corresponding location in the "fill_info_colours" vector. The vector index at which the point and colour are stored is returned to the caller of the operator. The image "picture" and, if "instant_update" is true, all displays that contain this image are updated to reflect this new fill information. The fill operation in effect here is a flood-fill of an interior defined 8-connected region. The region is defined by the input point, i.e. the region is all those pixels connected to the input point and of the same colour. The input colour specifies the colour that all these pixels should be changed to by the fill operation.

remove_fill_info - removes fill information from a given image. The index at which the fill information may be found in the "fill_info_points" and "fill_info_colours" vectors must be provided as input to this operator. The entries of these vectors at this index are removed and the vectors compressed. The "picture" of the image and, if "instant_update" is true, all affected displays are updated to reflect this fill information change.

3 Figure Transform Manipulation Operators

create_fig_trans - creates a new figure_transform instance. Two input parameters are required by this operator. The first, which specifies the "figure" of the figure_transform, may be a

graphical primitive or may be undefined. If it is undefined, the "figure" is defined as a vector of figure_transforms that is, initially, empty. The second parameter specifies the "transform" of the figure_transform. The new figure_transform instance is added to the "world.figure_set". Initially, this figure_transform may not be belong to the "contents" of any image.

destroy_fig_trans - destroys a specified figure_transform by removing it from all images and parent figure_transforms of which it is a part and then removing it from the "world.figure_set". This operator does not destroy the child figure_transforms of a composite figure_transform. It simply destroys their relationship as elements of a vector.

add_to_figure - adds a specified figure_transform to the "figure" attribute of another (parent) figure_transform. All images that contain this parent figure_transform and, if "instant_update" is true, all displays that contain these images are updated to reflect this change.

remove_from_figure - removes a specified figure_transform from the "figure" of another (parent) figure_transform. All images that contain this parent figure_transform and, if "instant_update" is true, all displays that contain these images are updated to reflect this change.

transform_figure - changes the "transform" attribute of a given figure_transform. The new "transform" value must be provided as input to the operator. All images that contain this figure_transform and, if "instant_update" is true, all displays that contain these images are updated to reflect this change.

copy - takes a figure_transform as input and creates a new figure_transform instance which is a copy of this input figure_transform. Initially, the new figure_transform may not belong to the "contents" of any image.

4 Input Device Manipulation Operators

select_image - sends input information from a specified input device to an image. When an enabled input_device is activated this operator is called with the input_device as argument. The operator determines which image the input is intended for, based on the point associated with the input_device, and then relays the input information to this image.

REFERENCES

[Acquah 1982] Acquah, J., Foley, J., Sibert, J., and Wenner, P., "A Conceptual Model of Raster Graphics Systems", Computer Graphics, Vol. 16, No. 3, July 1982.

[Arnold 1981] Arnold, D., "The Requirement for Process Structured Graphics Systems", Computer Graphics, Vol. 15, No. 2, July 1981.

[Ball 1983] Ball, J.E., "Canvas: the Spice Graphics Package", S108, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1983.

[BYTE 1981] "Smalltalk", BYTE, Vol. 6, No. 8, August 1981.

[Green 1981] Green, M., "A Specification Language and Design Notation for Graphical User Interfaces", TR 81-CS-09, Unit for Computer Science, McMaster University, Hamilton, Ontario, 1981.

[Green 1982a] Green, M., "A Design Philosophy for the Next Generation of Graphics Packages", unpublished manuscript, Unit for Computer Science, McMaster University, Hamilton, Ontario, 1982.

[Green 1982b] Green, M., "A Formal Study of the Window Interaction Technique", Unit for Computer Science, McMaster University, Hamilton, Ontario, 1982.

[Green and Philp 1982] Green, M., and Philp, P., "The Use of Object Oriented Languages in Graphics Programming", Proceedings Graphics Interface '82, May 1982.

[GSPC 1979] SIGGRAPH-ACM (GSPC), "Status Report of the Graphics Standards Planning Committee", Computer Graphics, Vol. 13, No. 3, August 1979.

[Foley and Van Dam 1982] Foley, J.D., and Van Dam, A., Fundamentals of Interactive Computer Graphics ,Addison-Wesley, 1982.

[Hindin 1984] Hindin, H.J., "Graphics Standards Finally Start to Sort Themselves Out", Computer Design, Vol. 23, No. 5, May 1984.

[Meads 1984] Meads, J.A., "The Graphics Standards Battle", Datamation, Vol. 30, No. 6, May 1984.

[Michener and Van Dam 1978] Michener, J.C., and Van Dam, A., "A Functional Overview of the Core System with Glossary", Computing Surveys, Vol. 10, No. 4, December 1978.

[Newman and Sproull 1979] Newman, W.M., and Sproull, R.F., Principles of Interactive Computer Graphics ,Second Edition, McGraw-Hill, 1979.

[Newman and Van Dam 1978] Newman, W.M., and Van Dam, A., "Recent Efforts Towards Graphics Standardization", Computing Surveys, Vol. 10, No. 4, December 1978.

[Robson 1981] Robson, D., "Object-Oriented Software Systems", BYTE, Vol. 6, No. 8, August 1981.

[Rosenthal 1981] Rosenthal, D.S.H., "Methodology in Computer Graphics Re-Examined", Computer Graphics, Vol. 15, No. 2, July 1981.

[Rosenthal 1982] Rosenthal, D.S.H., Michener, J.C., Pfaff, G., Kessener, R., and Sabin, M., "The Detailed Semantics of Graphics Input Devices", Computer Graphics, Vol. 16, No. 3, July 1982.

[Rosenthal 1983] Rosenthal, D.S.H., "Managing Graphical Resources", Computer Graphics, Vol. 17, No. 1, January 1983.

[Simons 1983] Simons, R.W., "Minimal GKS", Computer Graphics, Vol. 17, No. 3, July 1983.

[Wallace 1976] Wallace, V.L., "The Semantics of Graphics Input Devices", Computer Graphics, Vol. 10, No. 1, Spring 1976.

[Whitton 1984] Whitton, M.C., "Memory Design for Raster Graphics Displays", IEEE Computer Graphics and Applications, Vol. 4, No. 3, March 1984.

[X3H3 1984] ANSI (X3H3), "Special GKS Issue", Computer Graphics, February 1984.