A DEVICE-INDEPENDENT COMPUTER GRAPHICS LIBRARY

THE DEVELOPMENT OF A DEVICE-INDEPENDENT COMPUTER GRAPHICS LIBRARY BASED ON THE CORE SYSTEM

By

OWEN DOUGLAS FRANCES PLOWMAN, 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

April 1983

MASTER OF SCIENCE (1983)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: The Development of a Device-Independent Computer Graphics Library Based on the Core System

AUTHOR: Owen Douglas Frances Plowman, B.Sc. (Biology, McMaster University)

SUPERVISOR: Professor K.A. Redish

NUMBER OF PAGES: viii, 165

ABSTRACT

It has been recognized for some years that the use of computer graphics systems has great potential for improving man/computer communication. In the past, however, the high cost of graphics hardware, and the lack of accepted principles for graphics programming, prevented the widespread use of such systems. Recently, hardware has become more readily available, and efforts have been made to develop graphics software standards. This report presents an overview of one of the proposed standards, the Core System, and also discusses a portable subroutine library, based on the the Core System, that has been developed for use at McMaster University. This library, called SSOCS, is written in Pascal, and allows a user to produce two-dimensional images without regard to the characteristics of the graphics devices being used.

ACKNOWLEDGEMENTS

It is my pleasure to thank Prof. Ken Redish for his direction, encouragement, and patience (together with innumerable cups of coffee) during the course of this work. I would also like to thank the members of the Unit for Computer Science, and my fellow students in the Computation programme, for making my stay in the Unit very memorable.

I wish to express my appreciation to Dr. Heinz Klein, for his helpful discussions and critical comments, and also to Dr. Richard Welke, for his support over the last two years.

I am indebted to my friends for helping me to see this thing through; in particular, Cary, Patrick, and Pierre have given me invaluable support, in the bad times as well as the good.

Finally, my family has been behind me all the way on this project, and I dedicate it to them.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 What is Computer Graphics?

A basic definition of computer graphics is that it is the area of computer science which is concerned with the creation and manipulation of pictures. This kind of statement, though, is rather simplistic. The variety of pictures that can be generated by a computer graphics system range from simple two-dimensional plots and histograms, that might be produced by an average user, to complex dynamic three-dimensional images, used in aircraft simulators and the film industry. The field itself can be divided into passive (non-interactive) and interactive graphics. The former deals with the production of images on output devices, such as plotters and CRT-based displays, where an observer has little control over the appearance of the images. Interactive graphics, in contrast, is concerned with providing an observer with the ability to manipulate images through graphical means. Input devices such as light pens, joysticks, or digitizing tablets can be used to dynamically control the size, content, and format of the output on a display device.

1.2 The Importance of Computer Graphics

The introduction and proliferation of computer technology has had a revolutionary impact on our society, and we are becoming more and more dependent on the use of computers for the smooth functioning of our everyday affairs. This extraordinary influence is attributable to several factors, notably the great number of potential uses of computers, and the explosive growth of their capabilities, in terms of processing speed and capacity, together with their decreasing costs.

A consequence of these factors is that our expectations have changed. Computers used to be sufficiently complex and expensive that only fairly large organizations could afford to own or use them. They were initially used for scientific applications, dealing with numerical data, and then for commercial tasks, where the ability to store and process large amounts of character and numeric data was of primary concern. Today it is possible for individuals to own or use quite sophisticated 'personal' computer systems, and applications have spread to areas which do not necessarily have any _prima facie_ computational aspects (for example, visitors to major Canadian cities can use the Telidon system to obtain information on restaurants, tours, and entertainment).

This expansion of the use of computers does have some drawbacks. In the past, the usual means of interacting with a computer has been through some sort of textual medium: to produce certain results, people had to write or use programs in some computer language, and they 'used' the computer primarily through a keyboard on a card punch or a terminal. This kind of interaction may have been sufficient for tasks where some sort of end result was the focus of the application, but a different trend is now emerging. Because of the diversity of computer applications much more emphasis is being placed on man/computer communication. For most ordinary users, and some professionals, however, the complexity of the man/computer interface represents a formidable barrier, rather than an open door to communication [DOHE79].

Since the early work of Sutherland[1] in the 1960's, it has been clear that the use of computer graphics systems could greatly improve the quality of man/computer interaction: "The power, value, and success of Man/Computer communication via a good graphics system is undoubted" [STEW79]. The presentation of information and relationships in a pictorial format is a very effective means of communication, and it may be applied to a wide range of applications that do not have any obvious visual component. For example, incorporating a graphics display facility into a system that is essentially for business data-processing could have considerable benefit, since the information output from such a system would be enhanced by the expression of infor-

[1]Published in 1963, and referenced in [SUTH70].

mation in a visual form [STEW79]. The office environment is one area where this could have substantial impact, since management or clerical information could be more readily assimilated or interpreted when presented in a pictorial format. This kind of use of computer graphics is one of the motives behind the design of systems such as the Xerox Star and Apple LISA workstations.

Given the potential for improved man/computer communication, a reasonable question to ask at this point is why computer graphics technology is not more widely used. There are several plausible reasons for this. One is that the relevance of graphics to many applications is not always obvious (for example, office workers may be less likely to think in visual terms than people in creative disciplines like architecture and design [STEW79]). Another reason is that until recently computer graphics equipment was expensive, and investment in it could not always be justified. In the past few years, however, the 'microelectronic revolution' has led to a substantial increase in the sophistication of computer graphics hardware, with a decrease in cost, to the point where it is now readily available to the average user.

Graphics software has been, and remains, a major obstacle. The wide range of applications, the great number of equipment manufacturers, and the lack of an acceptable set of principles for graphics programming were a major source of discouragement to potential users in the 1970s, and delayed the acceptance of computer graphics systems [NEWM78]. In an attempt to overcome this problem, work was undertaken to develop a standard for programming graphics software. Such a standard, by specifying agreed-upon properties of graphics systems, would tend to increase software portability, and act as a foundation for the development of new applications.

One of the results of the movement towards standardization is the proposed Core System Graphics Standard, which was developed by the Graphics Standard Planning Committee (GSPC) of the ACM Special Interest Group on Graphics (SIGGRAPH). This standard is designed to be used for graphical systems dealing with line drawing (also called vector) graphics.

1.3 The Objective of this Project

At the present time at McMaster University, there are a number of passive graphical devices available for use, attached to different computer systems. A considerable need exists for software which will allow individuals to use these devices, without regard to their specific characteristics, or to the characteristics of the hardware to which they are attached.

The objective of this project was to construct a portable subroutine library, based on the Core System specification. This library, called SSOCS (Subset of Core System) is written in

Pascal, and implements the features of the Core System that are considered to be necessary for the development of passive graphics packages oriented towards the production of simple two-dimensional charts and graphs.

CHAPTER 2

THE STANDARDIZATION ISSUE

2.1 The Need for a Graphics Standard

In the early 1970s the lack of a set of commonly-accepted codes of practice for developing computer graphics systems became a source of concern for many workers in the field. Too often, the design and implementation of new applications was requiring a large amount of time and programming, and the results of this effort were not satisfactory. Newman and Sproull pointed out in 1974 that: "Virtually every time a graphical display terminal is attached to a computer ... a fresh graphics software system must be written to support it" [NEWM74]. The lack of understanding of what the important methodological principles of computer graphics were contributed to the problem.

One result of the attempts to resolve these difficulties was a movement towards the standardization of graphics systems. By making a standard available at all graphics installations, a uniform interface to graphics devices would be created. The following benefits could then be realized:

1. improved software portability, allowing graphics application software to be transported from one install-

ation to another with minimal changes [NEWM78] [GUED76].

2. improved 'programmer portability'; that is, lowering training costs and easing the task of a programmer, by decreasing the number of graphics systems with which it is necessary to be familiar [NEWM78].
3. improved productivity; applications based on a standard system, instead of one which is specially built, should be easier and faster to develop [GUED76].

2.2 The Seillac Workshop

To organize the standardization efforts, the Graphics Standards Planning Committee (GSPC) was formed in the United States in 1974, and IFIP WG5.2 was set up in Europe in 1975. Few tangible results appeared, though, until 1976. A key event in that year was a workshop which took place in Seillac, France, under the auspices of IFIP WG5.2. This workshop, called "Methodology in Computer Graphics" was held to allow the study of basic issues, and clarification of the underlying concepts of computer graphics, leading to a better understanding of its methodology. It was hoped that this would resolve some of the more difficult issues raised by the question of "what should be in the standard?", and that a set of generally agreed-upon principles of graphics system programming could be produced. The proposal for a standard that resulted from the workshop had a great influence on graphics standard development, and formed the basis for the design of the Core System and the Graphical Kernel System (GKS).[1]

One of the fundamental issues in the Seillac proposal deals with the relationship between graphical 'modelling' and 'viewing' functions. Modelling is concerned with the definition of graphical objects and the relations between them: it determines what is in the application program's world. Viewing functions are used to determine what information the end-user actually sees on a device: how the world appears. Since the former is more likely to be dependent on a particular application, it was proposed that a graphical standard should only be concerned with functions that are used for viewing purposes; any modelling capabilities that could be built on top of such functions should be excluded. This decision helped to focus attention on what graphical capabilities should be standardized.

A graphical standard should provide a set of functions which can be used to build descriptions of graphical entities. A fundamental strategy for achieving the portability desired in such a standard is the provision of features which shield the application programmer from specific hardware considerations [BØ80] [NEWM78]; this is termed 'device-independence'. With this goal in mind, various approaches to standardization are possible.

[1]A brief overview of GKS, and its relationship to the Core System, is presented in Appendix I.

The participants of the Seillac workshop considered proposing a standard graphics language; the lack of consensus about the important concepts, goals, and semantic facilities for such a language, however, showed that it was not an ideal solution. Extensions could have been made to existing programming languages, to allow the expression of graphical abstractions, but to adopt this approach would have meant modifying the compilers for whatever language was to be extended, and was not feasible. To standardize at a low level, oriented towards hardware, would be impractical, since equipment capabilities are very diverse, and a large interface would therefore be required. Conversely, standardization at the application level would be too specific and restrictive, since the standard should be appropriate for as wide a range of tasks as possible.

It was agreed that any proposed standard should be applicable at a programming language level [HOPG76]. The Seillac workshop attendees therefore recommended defining a standard subroutine library, which would be general enough to support a wide variety of users, and with an underlying conceptual model simple enough to be easily understood by programmers and users [ENCA76]. It was recognized that functions contained in the library should have a potential for long life, stability against change, and a high probability of representing the best techniques for graphical work [SANC76]. In order to produce some concrete proposals in a reasonable time, the following principles were used to establish the scope of the proposed standard:

1. functions for the construction and manipulation of 'pictures' should be provided;
2. 'pictures' should consist of text and vectors;
3. only graphical devices suitable for a general-purpose system should be included;
4. the graphical functions available should reflect the capabilities of the most frequently used devices, in order to ensure efficient use of these devices.

It became obvious, at the Seillac workshop, that a standard could not be based on any of the existing graphics software packages. While there appeared to be some consensus, on the part of the designers of this software, about what features were important for an effective graphics package, there was also a considerable diversity between the packages. In order to avoid constraining the potential applications of the standard, and the computer graphics equipment it could be implemented on, the Seillac workshop participants decided to include features that would be _generally_ useful, and which would not unnecessarily complicate the underlying programmer's model. After a good deal of discussion, it was agreed that the following modules should be included:

1. a limited set of output primitives, or lowest-level

graphical functions[1], for the specification of pictures. The set of primitives should be rich enough to use most of the capabilities of specific devices.

2. two- and three-dimensional viewing transformations, which would allow objects to be described in an application-dependent coordinate system, and then mapped to actual physical devices.
3. a limited set of interaction primitives, to allow selected virtual input devices to be used.
4. one level of segmentation; a segment is a named collection of graphical primitives that can be manipulated as a distinct unit.
5. functions to manipulate attributes of the graphical primitives, such as colour and style of lines.
6. control functions.

An important methodological concept that was uncovered at the Seillac workshop concerns the relationship of application program structure to the design of a standard. Since the behav-

---

[1]Foley [FOLE76] defines a primitive as:

1. a function which directly takes advantage of the hardware capabilities of commercially-available devices,
2. a function which could directly take advantage of the hardware characteristics of new devices which might become available,
3. a construct which cannot easily be simulated on existing or planned devices.

iour of an application program is often dependent on the actual graphics devices it employs, its underlying design may also be oriented towards these devices. This conflicts with the primary goal of standardization: to make systems more portable. An effective standard must, therefore, contain features that allow programs to be transported successfully with a minimum of change to their design. The functional capabilities inherent in the modular structure of the Seillac workshop's proposal provide a well-structured, device-independent interface to graphics equipment, and can be used to meet the needs of different types of applications.

CHAPTER 3

AN OVERVIEW OF THE CORE SYSTEM

The 1976 Seillac workshop represented a considerable step forward in the efforts to standardize graphics systems. As well as focussing attention on issues of graphics methodology, the attendees made a number of proposals regarding standardization. Following on from this work, the GSPC set out to produce a specification for a standard, based on the Seillac recommendations.

One of the most important concepts in the Seillac proposals concerns the modelling/viewing dichotomy. It was recommended that a graphical standard should only be concerned with viewing functions, which determine the information that is visible on a graphical device. These functions should be generally useful, not cost too much to implement, and not introduce hard-to-resolve issues [GSPC79]. The GSPC's specification was therefore viewed as a kind of 'core' of capabilities, which would have a variety of high-level application systems built around it, and it became known as the Core System (or Core). This system is oriented primarily towards applications using medium performance vector graphics displays. While it allows quite powerful line-oriented graphical manipulations, the necessity to remain both device-independent and useful for a variety of tasks means that it ignores more sophisticated capabilities that are found on some devices[1].

A preliminary document specifying the Core System was published in [GSPC77]. This document described a draft standard, intended to provoke discussion and further work, so that a more complete and useful version could be developed. A second attempt at a specification, based on comments and criticisms received in response to [GSPC77], was published in [GSPC79]. This chapter presents a summary of the major features of the Core System as it was described in that document. Raster Extensions and the GSPC Metafile Proposal are not discussed, since they are outside the scope of this project.

3.1 The Programmer's Model

The routines provided by the Core present the application programmer with a simple yet powerful conceptual model, based on the following ideas:

1. there are separate sets of functions for input and output. Together these functions provide a logical

[1]The Core does however, allow an installation to implement functions which take advantage of device capabilities it does not support directly (for example, curve generation on a display terminal). These kinds of functions are called 'escapes'. They are required to interface with the Core System in a well-defined, uniform way, and provide "a standard way of being non-standard" [GSPC79].

graphics system, independent of the graphical equipment which is actually available. There is little difference, for example, between using the Core to create an image on a plotter or a display terminal. Similarly, input functions can be used without knowing the specifics of actual input devices.

A description of an object to be displayed is built up through invocation of the output functions, which create instances of output primitives. The appearance of the output primitives depends on the values of primitive attributes.

2. there are two coordinate systems. The application programmer manipulates images in the 'world' coordinate system (which is application-dependent), while the actual information that is visible is in a normalized device coordinate system. Some mechanism must be available for mapping coordinates from one system to the other, and this is called the 'viewing transformation'. The application programmer can define the bounds of the world, and, using normalized device coordinates, where the picture of an object is to be placed on a logical output device. The Core System will handle the transformation of the object's world coordinate description to normalized device coord-

inates, and then to the coordinates of the actual physical device selected

3. all of the output primitives for an object are placed in segments. Each segment contains a part of the picture being displayed, and can be used to manipulate a collection of primitives as a distinct unit.

3.2 Output Primitive Functions

Output primitive functions are used to build descriptions of objects in the graphical world. Each invocation of an output primitive function produces a primitive instance that specifies one part of the total picture. The functions can be thought of as simulating the action of a plotter pen, or the beam of a storage tube device; there are six classes, describing moves (without producing any visible line), lines, connected sequences of lines, marker symbols, sequences of marker symbols, and text.

The output primitive functions all use the world coordinate system, and operate with reference to the 'current position'. This is an intrinsic Core System value which defines the current location of the simulated pen. There are equivalent functions in each class for specifying coordinates that are absolute or relative (to the current position). Also, since some applications require three-dimensional capabilities, as well as two-dimensional, they are incorporated into the Core System (this has an impact on the design of the Core's viewing specification, described in section 3.5).

3.3 Primitive Attributes

The characteristics of each output primitive instance depend on the values of attributes. These can be changed by the application programmer at any time, using appropriate functions.

Several attributes are applicable to all of the output primitives. These are colour, intensity, type of 'pen' used to draw the primitive, and its 'pick id'. Colour determines the colour of the output primitive, while intensity affects its relative brightness. The pen attribute is used to distinguish the image of a particular primitive: the different types of logical pen provided are installation-dependent and may employ several other attributes. For example, the default pen provided when the Core System is initialized might be one that draws black solid lines. With different logical pens available, the application programmer might be able to select one to draw red dashed lines. The pick id attribute is used to associate a name with an output primitive, to allow it to be selected by an end-user using a PICK input device (described in section 3.6).

Other attributes are applicable only to some of the output primitives. A 'marker-symbol' attribute defines the type of symbol to be used as a marker. The style and width of lines can be varied. Powerful capabilities exist for affecting the appearance of textual primitives. For example, the size, font, and orientation of characters in the world coordinate system can be manipulated by the application programmer.

3.4 Segments

Since the Core System is designed to allow interactive use of computer graphics devices, some capability has to be included to which allow the interactive modification of the images displayed. This is necessary, for example, to allow a portion of a picture selected by an operator to be deleted. The Core provides a graphical data structure called the 'segment', which groups output primitive instances together into separate named units. A picture is built up by successively creating a segment, invoking output primitive functions, and then closing the segment.

Segmentation gives a single level of partitioning of graphical data: each segment can only contain output primitives. An alternative method of structuring would have been to use a hierarchical approach; that is, to build a collection of units, each of which consists of primitives, or references to other units. While this is a very flexible way of partitioning an image, it was rejected for the Core System, since (a) there was no agreement on the best type of structure, and (b) it was felt that most applications would not need this flexibility [MICH78].

There are actually two kinds of segments available to an application programmer using the Core System. 'Temporary' segments are useful for purposes where data is to be displayed without modification, such as in a completely passive plotting package. To retain information in segments would be a needless overhead for such a package. None of the information placed in a temporary segment is recorded, so that if a display surface is cleared, any information placed in such a segment is lost. 'Retained' segments gather primitives into units which can be given names. To modify a displayed image, it is only necessary to change the segments which contain the primitives describing that part of the picture. "Changing", in this context, means that segments have to be deleted and recreated. The Core System does not provide a segment modification operation, since if a segment was closed at a certain point, and later on modification was required, it would have to be reopened in an identical state for additions or deletions of primitives to be made. This would require that the viewing parameters, which describe the graphical world, would have to be saved with each segment, and this was thought to be too expensive to implement. The designers of the Core also felt that most applications could be implemented without needing any modification or extension capability [MICH78].

The Core provides segment attributes, which can be used to change characteristics of retained segments (temporary segments have no attributes). Dynamic attributes can be changed at any time after a retained segment has been created, and affect properties such as its visibility (it may be visible or invisible), and size, position, and orientation (through an image transformation). A single static attribute determines whether an image transformation can be applied to a particular segment. There is no facility for sharing attributes between primitives and segments. For example, it is not possible to create a segment, populate it with primitives drawn in a certain colour, close the segment, and then later change the colours of the primitives in one step. Primitive attributes, therefore, affect the characteristics of an object, while segment attributes affect the displayed image of the object.

3.5 The Viewing Transformation

The role of the viewing transformation has already been briefly mentioned: it specifies the limits of the graphical world, and maps descriptions of objects defined in the world coordinate system to equivalent descriptions in device-dependent coordinates. The designers of the Core System used a conceptual model based on a 'synthetic camera' to develop the implementation of this transformation. In this analogy, the viewing transformation simulates a camera which takes snapshots of the graphical world.

Two- and three-dimensional viewing operations are combined in the Core; the former are considered to be special cases of the latter. If a three-dimensional viewing tranformation has been specified, both two- and three-dimensional output primitives can be used to describe objects. Two-dimensional primitives are simply considered to have a missing coordinate, and this is supplied by the current position value. The viewing operations, therefore, specify all necessary information for the placement of the synthetic camera; that is, its location, the orientation of the lens and film plane, and the vertical axis. A snapshot is begun when a segment is open, and is completed when the segment is closed. Because of this, the viewing transformation cannot be changed while a segment is open.

3.5.1 Two-dimensional Viewing

To describe a two-dimensional object an application program first specifies the bounds of a rectangular 'window'. This represents the portion of the two-dimensional world which contains the objects of interest, and it may be rotated about the origin of the world coordinate system axes. If part of an object being described by primitive functions falls outside the window, it may be 'clipped'; this process divides an image into visible and invisible portions, so that only the information inside the window (i.e. which is visible) is mapped to device coordinates. To accomplish this mapping, a 'viewport' is specified, on a logical 'view surface'. A view surface is a rectangular area on a selected device, and has dimensions which range from 0 to 1 along both the X- and Y-axes[1]; it therefore defines a normalized device coordinate space.

To produce a display, the Core System maps the contents of the window to the viewport, and then to the physical device surface, to give one particular view of the graphical world. Coordinates are therefore transformed from the world coordinate system, to normalized device coordinates, and then to actual physical device coordinates.

3.5.2 Three-dimensional Viewing

In order to display a three-dimensional picture on the surface of a two-dimensional device, the viewing transformation has to specify a projection from three dimensions to two, as well as clipping to a window and mapping to a view surface. The mapping from the two-dimensional projection to the view surface device coordinates is carried out in a similar way to two-dimensional mapping. The projection is accomplished in the following way. First, an application program sets up a view plane (also called a projection plane) within the world, and defines a portion of this plane to be the window. Objects in the world will be projected onto the view plane for display purposes. Next, the type of projection is selected, in order to set up a 'view vol-

[1]To allow efficient use of non-square displays, this range may be constrained along one axis, but not both.

ume'. A view volume is a three-dimensional area in the world coordinate system, within which an object of interest appears. If a perspective projection has been specified, this volume is a pyramid, while in parallel projections it is a parallelepiped. The projection of an object is found by passing lines through each point on the object and finding their intersection wih the view plane. To produce a perspective projection, these lines are considered to emanate from a point called the 'centre of projection', while for a parallel projection, all the lines are parallel to a 'projection direction'.

In three dimensional viewing two kinds of clipping may occur. The view volume may be restricted through the specification of front and back clipping planes. 'Depth clipping' will then discard parts of the picture which protrude from the front or the back of this restricted view volume. Clipping can also occur against the window, as in the two-dimensional transformation; this determines what is visible in the view plane.

3.5.3 Other Capabilities

Although modelling was not considered to be part of the Core System, the viewing transformation may be preceded by a 'world coordinate transformation', which would allow a modelling system built into an application program to work more efficiently. This kind of transformation allows objects in the world coordinate system to be scaled, rotated, and positioned, independently of the viewing parameters.

The Core System also provides for an 'image transformation'. This allows scaling, rotation, and translation of coordinates in normalized device coordinate space, and is analagous to transforming a photograph of an object by repositioning it on the page of a photo album [GSPC79].

3.6 Input Primitive Functions

The Core System's input primitive facilities provide a conceptual framework for the interaction of an end-user with an application program. A set of logical input devices, which simulate typical physical input devices, is available, making this interaction independent of actual physical input devices that are present.

Each logical input device is defined in terms of the type of data produced by the corresponding physical device, and the way in which that data is captured by the application program [GSPC79]. Before any device can be used, it must be explicitly enabled by the application program. There are two different models of data input provided in the Core: synchronous, or sample-based, and asynchronous, or event-based. Synchronous devices produce values which may be sampled by the application program; when data is requested from such a device, the application program will wait until that data becomes available, and will then continue executing. Asynchronous devices signal events to the application program. As soon as such a device is enabled it may be used to generate 'event reports', which contain data concerning the state of the device when the event took place. The event reports are placed in a first-in, first-out 'event queue', and they may be removed and processed by the application program at any point. These devices may therefore be used without interrupting program execution.

The logical devices, their corresponding physical devices, and the input model they employ, are shown in Figure 1.

3.7 Control

The Core System provides several routines which exert a global control over the generation of pictures. Basically these routines fall into two groups, concerned with: (a) control over multiple view surfaces, and (b) control of picture changes.

3.7.1 Multiple View Surfaces

While many applications are designed with one end-user and graphical display device in mind, there are situations where more than one device may be required simultaneously. For example, a hardcopy of a displayed image might be needed at the same time it is being generated on a terminal. The Core System there-

FIGURE 1:

THE CORE SYSTEM INPUT DEVICES

LOGICAL DEVICE	PHYSICAL DEVICE	INPUT MODEL
PICK	Light Pen	Asynchronous
KEYBOARD	Alphanumeric Keyboard	Asynchronous
BUTTON	Program Function Key	Asynchronous
STROKE	Tablet Stylus / Joystick	Asynchronous
LOCATOR[1]	Tablet Stylus / Joystick	Synchronous
VALUATOR	Potentiometer	Synchronous

[1]A LOCATOR device provides a single item of coordinate information, while a STROKE device provides a series of items

fore allows an image to be displayed on more than one device at the same time. A limitation of this is that the images that result could be different, since display devices might differ in their capabilities (for example, the colour attribute might not be supported on a plotter). A routine is therefore provided that determines the capabilities of any given device.

3.7.2 Control of Picture Changes

The Core System provides several routines which allow an application programmer to exert control over the manner in which changes are made to pictures; this may be useful for reasons of efficiency. Three types of control are allowed:

1. control over the visibility of segments. In order to achieve near-simultaneity of changes to pictures, an application program can prepare new retained segments, but keep them invisible. When all required segments have been generated, they can all be made visible at once, while the segments which were previously being displayed are made invisible (and can then be deleted as necessary). This process obviously imposes overhead on an application, since extra segments must be stored.

2. control over the sequencing of changes. An application program may generate changes to a displayed

image in batches; this means that some changes will be deferred, and will not take effect until a later time, when the batch is said to be over (some changes may occur immediately, depending on the capabilities of the physical view surface). This kind of capability would be useful when a storage-tube device is being employed, and multiple deletions are occuring: instead of deleting each segment and redrawing the image several times, the deletions can be deferred so that only one erase and redraw of the screen is necessary.

3. control over the immediacy of changes. The Core allows a number of changes to be grouped into a block, and then sent to a device; this means that some changes may be delayed, and then appear in a burst. This capability is useful when the connection between the application software and a graphical device imposes high overhead.

3.8 Levels of the Core System

Since one of the motives behind developing a standard for graphics software is that it should be useful for a wide range of applications, the Core System has been designed in 'levels'. The rationale behind this is explained by the following example: a purely passive plotting package would not need any capabilities for retained segments or interaction, so if it had to include them in order to conform to the standard, this would be quite wasteful.

Three classes of upward-compatible levels are specified by the Core. One class deals with output capabilites, one with input, and the third with the number of dimensions in the world coordinate system (two or three). Classification of capabilities on this basis means that there are a finite number of 'dialects' of the Core, and their relationships and dependencies are well-defined.

CHAPTER 4

AN OVERVIEW OF THE SSOCS LIBRARY

4.1 Purpose

The primary objective of this project was to develop a subroutine library (also called a 'package'), based on the capabilities described in the Core System specification, that could be used by a programmer to generate reasonably simple two-dimensional pictures on different output devices. The library was therefore required to implement the facilities of the Core that would allow it to be device-independent. A secondary goal was to write the routines in the library so that they could be transported to different machines with a minimum of changes.

4.2 Capabilities

Given the primary goal of the project, it was judged that, as a minimum, the following functional areas of the Core System should be implemented in the SSOCS library:

1. two-dimensional output primitive functions;
2. two-dimensional viewing functions;
3. capabilities for handling multiple view surfaces.

At present, the library incorporates nearly all of these features (time limitations prevented the implementation of the full set)[1]. Appendix II contains a list of the available SSOCS routines together with their functions. The library provides many of the features of an Output Level 1 (Basic Output) implementation of the Core System. Its capabilities are compared with this level in Figure 2.

4.3 The Choice of Programming Language

Since a portable implementation of the SSOCS library was required, a high-level, widely-available programming language had to be used. It is probably safe to say that there is no single language which is best suited to the task of programming a graphics package like SSOCS. The positive features of the language chosen will nearly always be balanced by some negatives. Recognizing this, the GSPC did not establish any rules concerning the choice of language in which to develop the Core System; implementations have been produced in FORTRAN [WARN78] [FOLE81], APL [FRIE79], and Pascal [NICO81] [STLU82]. At the time this project was begun, the only machine on which the SSOCS library could be developed was the McMaster University Cyber 170/730. Suitable languages available on this machine were FORTRAN and Pascal.

[1]There is no world coordinate transformation available in the viewing operations.

FIGURE 2:

A COMPARISON OF THE CORE SYSTEM

(OUTPUT LEVEL 1) AND THE SSOCS LIBRARY

FUNCTIONAL CAPABILITIES AVAILABLE	CORE SYSTEM (BASIC OUTPUT)	SSOCS LIBRARY
Output Primitives	Yes	Yes
Primitive Attributes	Yes	No
Viewing	Yes	Most
Control	Yes	Some
Temporary Segments	Yes	Yes

FORTRAN is one of the most widely-used languages for scientific programming. It was the first high-level language, and was introduced in the 1950s. Since that time it has undergone many revisions, but one of its main design goals has remained the achievement of high execution efficiency. As a result, its language structures are quite simple. This, however, proves to be a disadvantage. A programming language can be considered to provide both a conceptual framework for thinking about problems, and a means of expressing solutions to those problems in a concrete form. To be useful, therefore, a language should be based on a set of concepts that provides a clear and powerful means of developing algorithms, and that results in a notation which is natural to the problem. The lack of a powerful set of control facilities and data structures in FORTRAN makes the structured development of programs difficult. On the positive side, however, writing the SSOCS package in standard FORTRAN would have ensured a high degree of portability, because the language is available on many machines.

Pascal was designed in the early 1970s to meet two principal objectives. The first was to develop a language suitable for the teaching of a systematic method of programming. As a result it is a reasonably simple language, and at the same time it is quite powerful. Concepts important for a disciplined approach to programming are naturally reflected by Pascal's syntax. The second objective was to produce reliable and efficient (and machine-independent) implementations of the language; many have been developed, and the use of Pascal has now spread from the academic world to commercial organizations.

Pascal was used as the implementation language for the SSOCS library. The crucial difference between the two languages available, which led to the selection of Pascal, is their facility for data abstraction. An abstraction of an object is used in a sense that is removed from its concrete specification or representation. That is, it is viewed in terms of a subset of its actual attributes. In the graphical world, the objects of interest are pictures, made up of points, line segments, and text. In a system which is modelling the graphical world, some representation has to be found for the data items which correspond to these objects. The closer this representation to the real world, the easier the development of routines which deal with the data items would be, since the programmer's perception of the graphical world would be more precisely reflected. An ideal programming language would allow the spurious properties of objects, and their representation inside the machine, to be ignored.

Data abstraction facilities in FORTRAN are not very sophisticated. Integer, real, logical, and character variables may be used, while arrays allow data structures that are composed of related variables (i.e. of the same primitive type) to be specified. FORTRAN variable names are restricted to a maximum length of six characters. In Pascal, three basic classes of type are available:

1. scalar types are integer, real, char, and boolean; it is also possible to define 'enumeration' types. These allow a programmer to introduce a new type identifier, and to specify the set of identifiers denoting the values of this new type.

2. structured types are the array, where all components must be of the same type, and the record, which allows components to be of different types.

3. pointer types, which allow a programmer to specify data objects that may change in size during the execution of a program.

Variable names in Pascal are not limited to any specific length, although standard Pascal only distinguishes the first eight characters of a name.[1]

These features allow a Pascal program to use data abstractions that are much closer to the real world than their counterparts in FORTRAN. Since a programmer can define new types, meaningful specifications for graphical objects can be

[1]In fact, the length is restricted to the maximum line length of a program, since identifiers cannot be continued over line boundaries.

developed, and this was judged to be an important capability, both for the SSOCS library, and for application programs using it. To be most effective the behaviour of the data objects introduced by an abstraction should be completely expressed in terms of a set of operations that are meaningful for those objects [LISK77]. It should be possible to specify every action to be carried out by way of these operations, which should also be the sole means of manipulating the data objects. Both FORTRAN and Pascal allow the specification of such a set of operations through the development of appropriate procedures and functions.

4.4 The Device-Independent/Device-Dependent Interface

The connection between the SSOCS package and the physical output devices that can be used by it is provided through the device driver routines. The capabilities of these routines are defined by the Device-Independent/Device-Dependent (DI/DD) interface. This interface has to be uniform for all devices, regardless of their capabilities, and at the same time it must allow future expansion of the device-independent portion of the package, as extra facilities are added. For example, the structure of the interface should not preclude the addition, at some later time, of capabilities for manipulating primitive attributes. One of the first tasks in developing the package was deciding on the level of the interface: too low a level, and inefficient use of devices could result, while too high a level could make the addition of drivers for different devices quite difficult. If the interface were to incorporate capabilities that are not supported directly by a display device, these features would have to be simulated by the driver routines for that device.

The SSOCS DI/DD interface was designed to reflect the capabilities commonly found on the display devices available at the time the library was written (for example, it incorporates a low-quality text-handling capability). Since the specification of the output primitives for the Core System also reflects this, the interface essentially provides the same conceptual graphics device model as the high-level routines available to an application program. As each output primitive is created it is converted to a normalized device coordinate representation by the device-independent portion of the SSOCS package, and then passed across the DI/DD interface to the device drivers, where it is transformed to an equivalent representation for selected devices.

A distinction is made, at the DI/DD interface, between producing graphical output and describing the state of the device environment on which that output is to appear. One pathway is provided for commands which actually perform output, and another for sending commands to a device to alter its properties. At present, the capabilities for altering the device-dependent graphical environments are restricted to accessing specific devices, preparing the device surfaces for graphical output, clearing the surfaces, and terminating use of the devices. Each alteration of the environment uses the command pathway to invoke a device driver routine, supplying it with the action to be taken, and the device to be affected. The routine simply transmits appropriate commands to the selected device to enable it to perform the required action. For graphical output, a set of routines is provided at the driver end of the output pathway; these perform specific primitive actions on each device.

4.5 Device-Independent Structure

4.5.1 Output Primitive Functions

The output primitive functions available in the SSOCS library allow a programmer to specify moves, lines, text, marker symbols, and sequences of lines and marker symbols in the world coordinate system. Either relative or absolute world coordinates may be used to define the locations of the primitive instances, by using the appropriate version of the functions. Each invocation of a function, except for a move, creates an instance of a data type called a primitive, which is implemented as a Pascal variant record; this structure can be used to represent any of the output primitives available, and carries the information specifying the action to be taken (line, marker symbol, etc.), and its coordinate information. The operations that can be carried out on a variable of type primitive are defined as low-level procedures and functions; this provides a way of manipulating the variable from other higher-level routines that is independent of its definition, and which is more natural[1]. There is no variable created for a move function: a move simply sets the current position in the world coordinate system, and this will not necessarily have any visible effect on a device. The other primitives record their starting and ending coordinates in their own primitive variables.

4.5.2 Viewing Operations

Information about the graphical world is captured in a data structure of type environment, which is a Pascal record used to store the window, viewport, and the size of the normalized device coordinate space. This information may be supplied by the application program, or set to default values. Like the type primitive, the representation of environment is 'hidden' from the remainder of the package: to describe the graphical world from high-level SSOCS routines, low level procedures and functions are used.

The viewing operations in the SSOCS package are implemented in a manner similar to that of the viewing 'pipeline' concept

[1]For example, a procedure 'set_action_to' determines what kind of primitive is currently being described (line, move, etc.); without this facility it would be necessary to refer to the actual structure of the data type in each place it used.

discussed in [FOLE81]. At the beginning of the pipeline, the invocation of an output primitive function creates an instance of a primitive. At this stage, the information contained in this is in a device-independent form. The primitive instance is sent along the viewing pipeline, and is prepared for display by transforming it into successively more device-dependent representations. The process is illustrated in Figure 3.

First, the output primitive is mapped to the window, to determine what portions of it are visible. The window, defined in world coordinate space, corresponds to the viewport, defined in normalized device coordinate space. Any primitives which appear inside the window will be visible on a display surface. Since the window may be rotated about the world origin by the application program, some representation of the effect of this is needed. In the SSOCS library, a physical rotation of the window is not actually performed. Instead, each coordinate pair is rotated, in the reverse direction, by the angle of rotation specified for the window. This process achieves the same effect as rotating the window. It simplifies the clipping stage of the viewing process, since the sides of the window are kept parallel to the coordinate axes.

In order to determine what elements of the graphical world are to be displayed on a device surface, each output primitive may undergo clipping. This process discards primitives, and parts of primitives, that do not lie inside the window, so

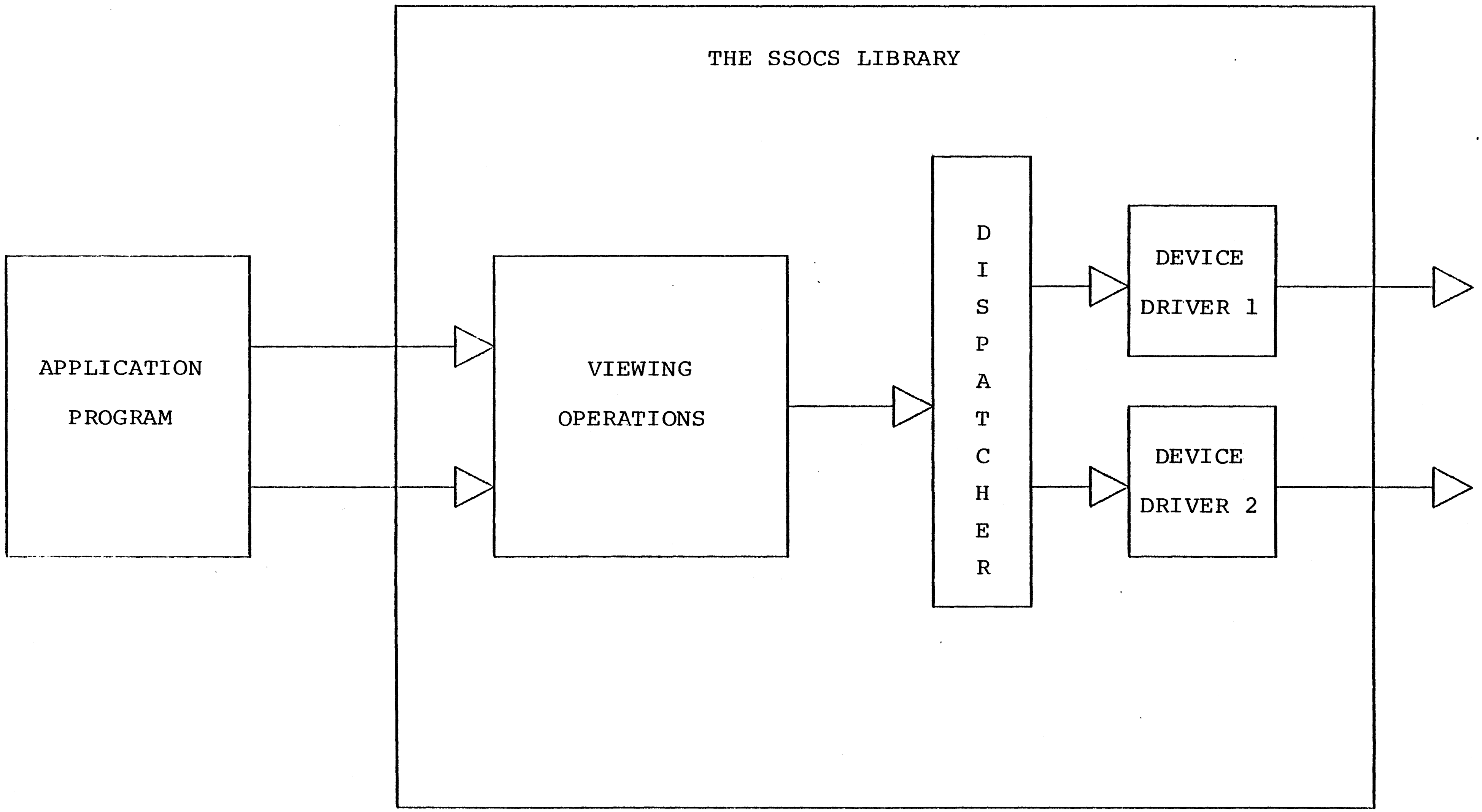

FIGURE 3: The SSOCS Viewing Operations

that only the picture elements that do appear inside are displayed on a device surface. Any primitive that lies partially inside and partially outside is cut off at the window edge before being transformed to normalized device coordinate space. This effect is illustrated in Figure 4. The SSOCS package uses a clipping algorithm developed by Cohen and Sutherland and described in [NEWM79].

The application program may request that clipping be turned off. In this situation, the window will still be mapped to the viewport, but if there are parts of a picture which lie outside the window, device-dependent results such as wraparound may occur. The ability to turn off the clipping is of practical importance since it will increase efficiency in applications where all primitives are known to lie within the window.

If the output primitive is discarded at the clipping stage of the viewing pipeline, further viewing operations will be abandoned, and control will return to the application program. If this does not occur, the primitive will be mapped to the current viewport, by transforming it to normalized device coordinate space. The output primitive is then passed to a 'dispatcher' module. This determines the kind of primitive that is to be displayed, and passes the primitive information down the DI/DD output pathway to the appropriate device driver routines.

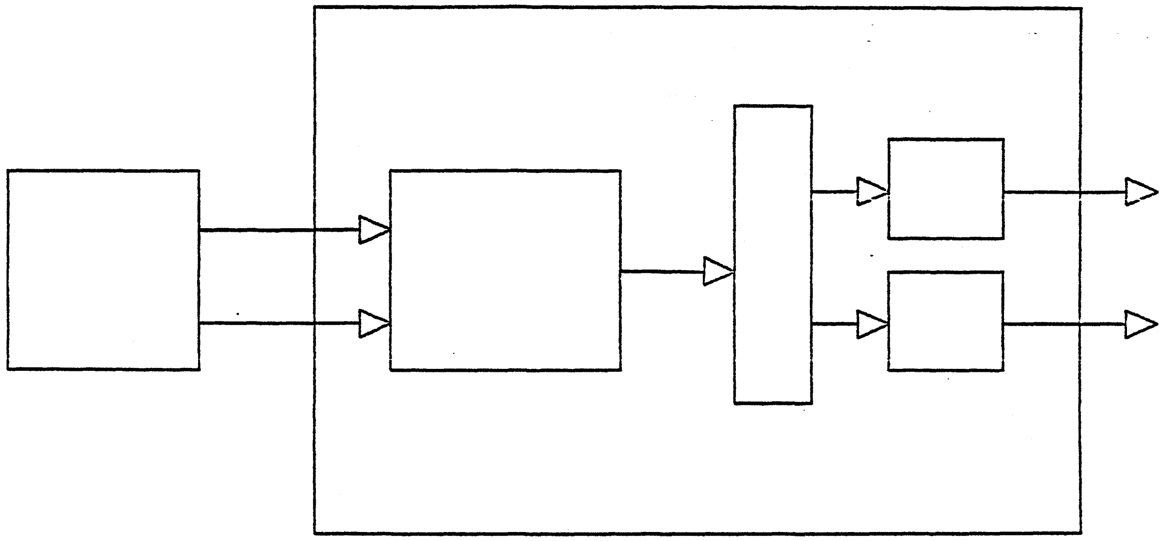

Fig. 4 (a): The original diagram of Fig. 3.

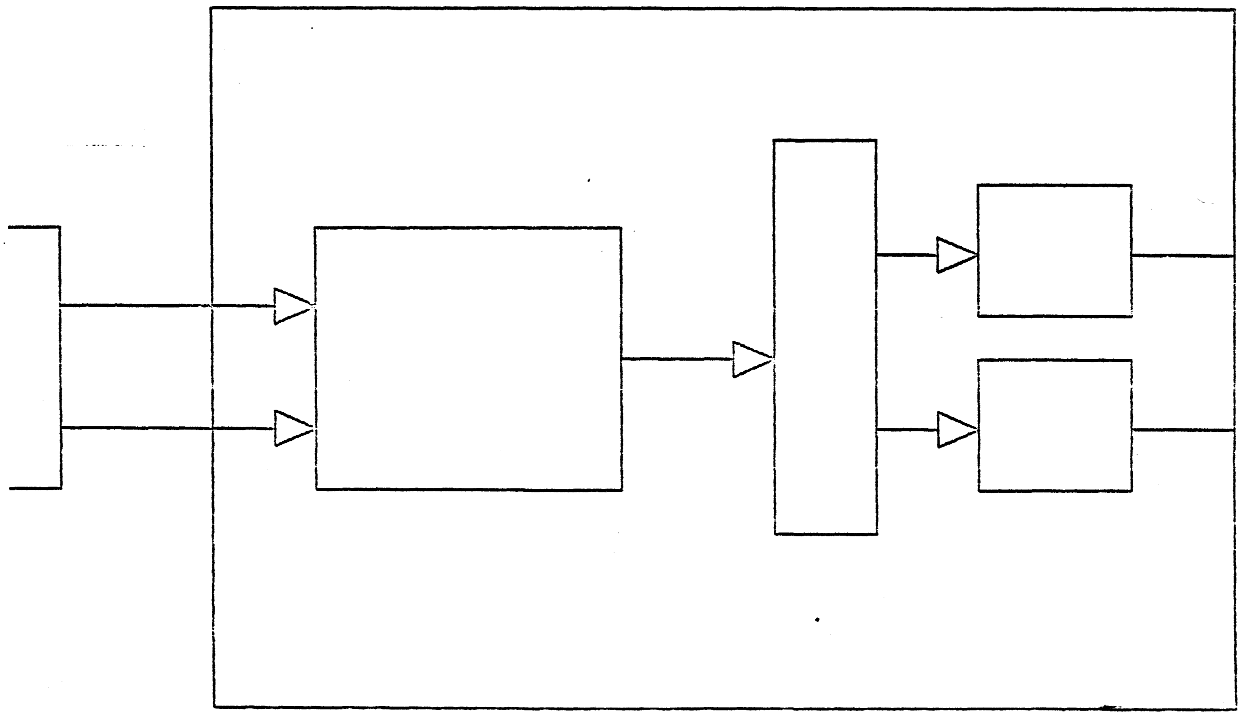

Fig. 4 (b): The window has been reduced in size, enlarging the diagram and clipping portions which lie outside.

FIGURE 4: The Effect of Clipping

4.5.3 Control Facilities

The SSOCS library allows output primitives to be displayed on more than one device surface at the same time. Each surface must be initialized before it can be used, and then explicitly selected before graphics output appears on it. A device cannot be released from output while a segment is open, and when a device is released the image displayed on its surface will remain until use of the device is terminated by the application program.

The library also provides the capability of forcing a device surface to be cleared, by requesting a 'new frame action'. This action will clear the screen of a CRT-based display, and will advance the paper on a hardcopy device. For some buffered devices (for example, plotters) images will not appear on the device view surface until this function is used, or the device is released and terminated.

Two routines control the initialization and termination of the entire SSOCS library. The initialization routine sets control variables to required defaults, and also assigns some of the default values for the device-independent graphical world (for example, the dimensions of the world coordinate system's window). Although the Core System specification document states that this routine is to be called before any others in the library, an application program using the SSOCS package must call another routine before this, in order to guarantee the initial states of control variables.[1] The SSOCS termination routine closes any segments that are open, and clears and releases any devices being used.

The SSOCS library provides similar error handling capabilities to those described in the Core System specification document, except that there is no provision for a programmer to define an application-dependent error handling routine[2]. Whenever an error is detected an error report is created, consisting of an error identification number and a code indicating the severity of the error. The report is then sent to whatever error log device is available. The most recent error report is maintained by the library for interrogation by the application program. The error indentification numbers, and severity codes, are listed at the end of Appendix II.

4.5.4 Escape Facilities

One escape routine is provided in the SSOCS package. This is used for creating circles on devices that possess a circle generator. The application program is required to specify the desired view surface, and the location of the circle on that

[1]This is necessary because Pascal does not allow variables to be initialized at compile time.

[2]The reason for this is that the library routines are loaded as a single unit at run time.

surface in normalized device coordinates. This information is passed directly to the dispatcher, which then invokes the appropriate device driver routines.

CHAPTER 5

THE SSOCS IMPLEMENTATION

The SSOCS library currently consists of about 3 000 lines of Pascal; this represents about 170 routines. Many of these routines are concerned with the data abstraction mechanism of the library. There are three main data structures. Two have already been mentioned in the previous chapter: one is used to represent an output primitive, and the other is used to hold information about the graphical world. The third structure is used for control, and is discussed in section 5.2. Once the important data structures were designed, and the routines for their manipulation coded, development of the library proceeded in a 'top down' manner. Each of the library's application program interface routines is derived from the corresponding routine description given in [GSPC79]. These routines were written and tested first, and lower-level routines were then incorporated. One of the principal considerations in the development of the routines was that the resulting library should be upwards compatible with the Core System, so that, over time, functions could be added to achieve an implementation closer to the graphics standard. This meant that any constraints imposed on the library by data structures and the hierarchy of routines had to be kept to a minimum.

5.1 Portability

Development of the SSOCS library was initially carried out on the McMaster Cyber computer using Pascal 6000. A working version of the library was eventually produced, with drivers for two quite different devices: a DEC GIGI terminal and a Versatec plotter. At this point the entire package was transferred to a Pixel 100/AP microcomputer (a 68000-based machine running under the UNIX operating system). A Pascal compiler produced by Silicon Valley Software (SVS) is available on this machine. At present most of the continuing development effort of the SSOCS library is being carried out on the Pixel implementation.

When the library was transferred to the Pixel only one statement had to be changed, in the type declaration section[1], to allow the device-independent portion of the library to compile correctly. Since the GIGI is very similar to the DEC VT125 graphics terminal that is available on the Pixel, minor modifications were made to the GIGI driver routines, and to the SSOCS library's interface to the operating system, and a second working version of the entire package was produced. One of the objectives of the design of the library -- to make it <u>transportable</u> -- has therefore been met.

---

[1]This change is discussed in section 5.3.

5.2 Internal Control

Since the application program interface routines are based on descriptions contained in the Core System specification, the question arises of how to deal with the SSOCS library internal control variables. These variables are required for routines to determine the state of the library (whether initialized or terminated), window, and viewport, whether normalized device coordinate space has been defined, if a segment is open or closed, whether an error is being processed, and so forth. Each of the Core System high-level descriptions includes the routine name, required parameters, an overview of the routine's purpose, and possible error messages that could result from an invocation of the routine. There is no provision in the parameter lists for passing control information between routines. Since there is no facility in Pascal for sharing data between two routines except by making this data global in a common ancestor of the routines[1], all control information has to be global to the library, and also to the application program. In order to hide as much of this information from the routines that do not require it as possible, the Pascal record structure shown in Figure 5 was defined. This structure allows the creation of a global library control package. By defining routines which operate on this package, and only accessing the control variables through these routines, the

[1]There is no counterpart to the labelled COMMON data areas available in FORTRAN.

```
type

   operations  = (up, down);
   condition   = (ouvert, ferme);
   switch      = (on, off);

   status_data =  record
                     core_state,
                     window_state,
                     vport_state,
                     ndc_state,
                     ndc_def_state,    : operations;
                     seg_state         : condition;
                     error_processing  : switch;
                     report_gone       : boolean
                  end;
```

FIGURE 5

THE SSOCS LIBRARY CONTROL VARIABLE

number of global control structures is reduced to one, and this is only manipulated through a well-defined interface.

5.3 Text-Handling Mechanism

The current version of the SSOCS library does not support primitive attributes. This causes difficulties in dealing with the text primitive described in the Core System specification, since attributes are used to determine the size, quality, orientation, and spacing of the output performed by this primitive. The SSOCS library, therefore, only supports the creation of low quality text. Furthermore, all characters which appear in an image must be the same size. The driver routines for each device are responsible for defining the size of each character.

The primitive function for producing textual output described in the Core System specification takes a character string as its argument. A serious limitation of standard Pascal is that its capabilities for the manipulation of character strings are not very elaborate. There is no predefined string type. The primary mechanism for storing a sequence of characters is to use an array of type char. This is not very satisfactory, however, since the size of a Pascal array is part of its type; general-purpose routines for processing strings of different sizes therefore cannot be written. All strings have to be made the same size, by padding with blanks, or some special string terminating character has to be inserted into the array.

In order to implement the text primitive in the SSOCS library, the non-standard string facilities provided by both Pascal 6000 and SVS Pascal were used. Pascal 6000 includes a facility for defining dynamic arrays, while SVS Pascal provides quite extensive string-handling features. In order to transfer the library from one machine to another a single change had to be made in the definition of the character string type. Although this departure from standard Pascal introduces an element of machine-dependence to the SSOCS library, it was judged to be worthwhile. An application programmer who wishes to perform some textual output uses a routine call of the following form:

```
text_string ('this is a message');
```

If the non-standard string-handling capabilities were not used the programmer would have to convert the string 'this is a message' to whatever type has been defined for strings in the library, padding with blanks or using termination characters.

5.4 Application Program Interface

The interface to the application program is different for the Cyber and Pixel versions of the SSOCS library. Standard Pascal does not allow the separate compilation of any routines. This means that the entire SSOCS library, in source form, would have to be incorporated into each application program if no machine-dependencies were allowed.[1] Pascal 6000 and SVS Pascal do support the creation of libraries of routines, but Pascal 6000 will not allow global variables to be referenced in the library. Since every SSOCS high-level routine references (at a minimum) the initialization status variable, contained in the global control package, use of the separate compilation facility on the Cyber is limited. SVS Pascal, however, treats independent compilation in a manner similar to UCSD Pascal. Routines to be compiled separately are placed in a module called a unit, which is a collection of declarations and statements with some portions accessible by other program units, and some private. Each unit contains an interface section, which describes what is available

[1]In contrast, FORTRAN does provide for separate compilation.

to other units, and an implementation section, which supplies the code required for the unit. The interface contains definitions of global constants, types, and variables, which can be used by other units.

Since the routines on the Cyber cannot actually be placed in a library that is external to the application program, all statements making up the library must be included in the program. Pascal 6000 provides a useful 'include' facility, which allows external text to be brought into a program and compiled, and the Cyber SSOCS library is incorporated in this way. On the Pixel, the application programmer inserts a statement into his program which declares that it uses the SSOCS library unit. Using this capability can decrease the time required to compile application programs by up to 30%. Although a significant amount of time is saved at the code translation stage, an increase in the time required to link-edit the program is apparent.

5.5 Device Drivers

5.5.1 Structure

The structure of the device drivers in the SSOCS library is quite simple. The state, or environment, of each device is determined by a single device control routine, MAKE_DEVICE, and each device possesses a set of routines which implement the SSOCS primitive actions at the device level, In the current SSOCS implementation the actions that may be performed by the control routine are:

1. accessing the physical device, which normally involves opening and rewinding a file to which the device is attached[1];
2. preparing the device for graphical output, which requires that the device display surface is cleared[2], and the device is made ready to receive graphics commands;
3. clearing the device's display surface;
4. terminating access to the device, which normally involves resetting the device to its default mode, and closing the file with which it is associated.

For some kinds of devices one or more of these actions will not be required. For example, in the current CYBER implementation, nothing has to be done either to access the VERSATEC plotter, or to prepare it to receive graphic commands. In contrast, to use the VT125 terminal with the Pixel version of SSOCS, the file associated with this terminal must be opened and rewound. To prepare the terminal for graphics output, MAKE_DEVICE writes to this file commands which (a) home the alphanumeric

[1]The way in which the files are connected to the devices is system-dependent.

[2]Erasing the screen on a terminal, and advancing the paper on the plotter.

cursor, (b) clear the alphanumeric screen, (c) turn on graphics mode, (d) clear the graphics screen, and (e) set the graphics origin position. The appropriate strings corresponding to the required commands are provided as Pascal constants.

Device-specific procedures are required to perform the following primitive actions:

1. draw a line, given the end points in normalized device coordinates;
2. draw a marker symbol at a point specified in normalized device coordinates;
3. write a text string beginning at a point specified in normalized device coordinates;
4. create a circle of given radius centred at a point specified in normalized device coordinates.

Each procedure has to transform the normalized device coordinates specified into device coordinates, and output the appropriate graphics command to the device. If a device cannot directly achieve one of these functions (for example, creating a circle) the procedure must simulate the function.

5.5.2 Current Device Drivers

The device drivers for the GIGI and VT125 terminals are very similar. To perform graphics output on these devices, a language called Regis (Remote Graphics Instruction Set) is used.

Regis consists of a set of commands, and options for those commands. Each command, or sequence of commands, sent to the terminal is preceded by a 'device control string' (an escape character, followed by 'Pp'); this causes the terminal to enter graphics mode. To terminate a sequence of Regis commands a termination string is used (escape followed by '/').

When one of these devices is initialized by the application program the library will rewind the appropriate file, and write the device control string to it. The device driver routines then translate the commands from the device-independent section into equivalent Regis commands, and write these commands to the file. In response to the routine call 'line_abs (1.0, 1.0)', for example, the VT125 driver could write the characters 'v [479, 479]' to the appropriate file. When a device is terminated the Regis termination string is sent to the file.

In order to use the Versatec plotter, which is attached to the Cyber, the device driver routines call FORTRAN routines in the Versatec plotting library, PLOTVER. These routines are declared to be 'external' to the library. No special action is taken when the plotter is initialized, but no output appears until the application program terminates the device. The driver then signals that the current plot is complete.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

Both of the objectives of this project have been met in the current implementation of the SSOCS library:

1. An application programmer who understands the conceptual model underlying the Core System can use the library to generate output on different graphics devices with little knowledge of the capabilities of the actual devices.[1]

2. The library is transportable. Two versions exist: one on a mainframe (Cyber), and one on a microcomputer (Pixel). Very little source code modification was needed to produce the second version. Also, drivers have been written for two quite different devices (a Versatec plotter and GIGI/VT125 terminal). This demonstrates that the task of producing a device driver for an additional device is reasonably straightforward, provided that the device

[1]Many of the figures in this report were produced using the Cyber version of SSOCS.

possesses the basic graphical capabilities required by the DI/DD interface.

Although it does provide device-independent graphical capabilities SSOCS is somewhat limited. It includes only the most basic features of the Core System that are needed for two-dimensional output. The most serious deficiency is the lack of primitive attributes. This restricts the ability of an application programmer to produce textual output, and also prevents the use of colour, varying style of line, etc.

6.2 Recommendations

Adding the Core System's primitive attribute manipulation capabilities to the SSOCS library should be of high priority. It would also be useful to add the features that allow an application program to use retained segments and segment attributes, since at present any capability for selectively modifying portions of a picture has to be built into the application program.

APPENDIX I

THE GRAPHICAL KERNEL SYSTEM (GKS)

GKS was developed by the German Institute for Standardization (DIN); like the Core System it is based on the Seillac workshop proposal for a graphic standard. Because it appears to possess some advantages over the Core System it has been proposed as an international graphics standard. The most important features of GKS are described here, and are discussed in detail in [PRES80] and [ENCA80].

For output, GKS provides basic line drawing primitives, as well as raster graphics primitives. Five classes of input device are supported. Unlike the Core System, GKS only deals with two-dimensional graphical capabilities, since the majority of graphics applications today are of a two-dimensional nature [BØ80]. It therefore presents a simpler conceptual model to the application programmer. The GKS viewing transformation is similar to the Core's two-dimensional viewing transformation. A window is used to determine the portion of the world coordinate system that is visible. This area is mapped to the viewport, which is defined in normalized device coordinates on a view surface.

GKS allows a picture to be structured into segments, which provide a selective modification facility. In contrast to the Core System, which allows segments to be composed only of output primitives, GKS segments may be composed of primitives or other segments, so that libraries of segments can be developed.

A basic concept in the system is the 'abstract graphics workstation'. This represents a collection of graphical devices, operated by an end-user, and treated as one logical unit for input and output. The use of the workstation provides a flexible way of adapting application program requirements to specific hardware devices.

Each implementation of GKS maintains a table describing the capabilities of the available workstations. A 'workspace transformation' is used to map the normalized device coordinate representation of an object to an equivalent representation in the device coordinate system of each workstation selected for output. The transformation may be set individually for each workstation, allowing the same output to appear on different devices in different scales. GKS also provides a set of workstation attributes. An application program can use these attributes to control the appearance of identical output primitives on different workstations.

APPENDIX II

THE SSOCS PROCEDURES

1. Type Declarations

Declarations of the types of parameters used by the SSOCS procedures are shown below:

```
const

   maxpoints  = 80;

type

   coordrange = (1 .. maxpoints);
   coordseq   = array [coordrange] of real;

   condition  = (ouvert, ferme);

   switch     = (on, off);

   devices    = (vt125, versatec, gigi);

   error_info = (identifier, code);
   error_data = array [error_info] of integer;
```

2. The Procedures

This section lists the procedures contained in the SSOCS library that may be called by an application program. Further information on the procedures can be found in [GSPC79]. Each entry here shows:

1. an example of the invocation of the procedure, giving the number and type of its parameters;
2. a description of the procedure's function, and the way it is implemented within the SSOCS library;
3. error messages that may be produced by invocation of the procedure.[1]

The procedures listed here are from the Pixel implementation of the SSOCS library. The Cyber version routines are similar, except that underscores ('_') are omitted.

2.1 MOVE_ABS (X, Y : REAL);

This procedure sets the current pen position to the point specified by (X, Y), where X and Y are absolute coordinates in the world coordinate system. The current position is an internal SSOCS value, and invocation of this routine does not cause any change to images on any selected view surface.

Errors: none.

2.2 MOVE_REL (DX, DY : REAL);

This procedure sets the current pen position to a new position DX and DY away from its present value. DX and DY are

[1]All of the procedures will report error 717 if INITIALIZE_CORE has not been called prior to their invocation.

relative coordinates in the world coordinate system. Like MOVE_ABS, MOVE_REL does not cause any change to images being displayed.

Errors: none.

2.3 QUERY_CP (VAR X, Y : REAL);

This procedure assigns the coordinates of the current pen position to the parameters X and Y.

Errors: none.

2.4 LINE_ABS (X, Y : REAL);

This procedure draws a line from the current pen position to the point specified by (X, Y), where X and Y are absolute coordinates in the world coordinate system. The point (X, Y) becomes the current pen position.

The procedure first determines whether the point specified by X and Y is coincident with the CP; if so, no further action is taken and the procedure terminates. If these points are distinct, LINE_ABS calls the low-level routines SET_ACTION_TO and SET_ENDPOINTS, to place the required line information in the primitive information packet. This packet is then sent down the viewing pipeline to the procedure SEND_TO_DISPLAY.

SEND_TO_DISPLAY performs the following:

1. mapping of the primitive to the current window through MAP_TO_WINDOW, which may involve (a) clipping (procedure CLIP), and (b) rotation of the points specifying the primitive (procedure ROTATE_POINT);
2. if the primitive is still visible after the above step, it is mapped to the current viewport (procedure MAP_TO_VIEWPORT), which involves transforming the world coordinate specification to normalized device coordinates;
3. the primitive is then passed to a procedure called DISPATCHER, which (a) determines the type of primitive, and (b) calls the appropriate device driver routines. The device driver routines transform the normalized device coordinates to device coordinates, and produce an image on selected device surfaces.

Errors: 201 (There is no open segment)

2.5 LINE_REL (DX, DY : REAL);

This procedure draws a line from the current pen position to a point DX and DY away from this position. DX and DY are relative coordinates in the world coordinate system. The point (DX, DY) becomes the current pen position.

LINE_REL converts the relative coordinates DX and DY to absolute, and then invokes LINE_ABS.

Errors: 201 (There is no open segment)

2.6 POLY_ABS (X_ARRAY, Y_ARRAY : COORDSEQ; N : COORDRANGE);

This procedure draws a sequence of lines. The sequence begins at the current pen position, runs to (X_ARRAY (1), Y_ARRAY (1)), and ends at (X_ARRAY (N), Y_ARRAY (N)). Coordinates contained in X_ARRAY and Y_ARRAY are absolute world coordinates. The current pen position is set to the point specified by (X_ARRAY (N), Y_ARRAY (N)).

POLY_ABS performs N invocations of LINE_ABS, supplying the appropriate coordinate pairs each time.

Errors: 2 (N is less than or equal to zero)
201 (There is no open segment)

2.7 POLY_REL(DX_ARRAY, DY_ARRAY: COORDSEQ; N: COORDRANGE);

This procedure draws a sequence of lines. The sequence begins at the current pen position, runs to (DX_ARRAY (1), DY_ARRAY (1)), and ends at (DX_ARRAY (N), DY_ARRAY (N)). Coordinates contained in DX_ARRAY and DY_ARRAY are relative world coordinates. The current pen position is set to the point specified by (DX_ARRAY (N), DY_ARRAY (N)).

POLY_REL performs N invocations of LINE_ABS, converting the relative coordinates to absolute each time.

Errors: 2 (N is less than or equal to zero)
201 (There is no open segment)

2.8 MARKER_ABS (X, Y : REAL);

This procedure sets the current pen position to the point specified by (X, Y), and creates a marker symbol (a dot) at that point. X and Y are <u>absolute</u> world coordinates.

MARKER_ABS essentially performs the same actions as LINE_ABS. The low-level routines SET_ACTION_TO and GO_TO_ACTION_POINT are used to place the required marker information in the primitive information packet, which is sent to the procedure SEND_TO_DISPLAY.

Errors: 201 (There is no open segment)

2.9 MARKER_REL (DX, DY : REAL);

This procedure sets the current pen position to a point DX and DY away, and creates a marker symbol (a dot) at that point. DX and DY are <u>relative</u> world coordinates.

MARKER_REL converts the relative coordinates DX and DY to absolute, and then calls MARKER_ABS.

Errors: 201 (There is no open segment)

2.10 CREATE_TEMPORARY_SEGMENT;

This routine creates a new, empty, temporary segment, which becomes the open segment. Invocations of output primitive routines that create output primitives will result in new information appearing on selected view surfaces.

While a segment is open, the viewing specification may not be changed, and view surfaces may not be selected or deselected.

This procedure checks that at least one view surface has been selected for output, and that a segment has not already been opened. If both these conditions are satisfied, the SSOCS control variable is set to indicate that a segment has been opened, and CHECK_NDC_SPACE_STATE is then called to determine the state of the normalized device coordinate space.

CHECK_NDC_SPACE_STATE sets the NDC space to its default values (procedure DEFINE_NDC_SPACE), if it has not already been defined by the application program. It then creates the viewport to default specifications (procedure DEFINE_VIEWPORT), if this has not already been defined, and will determine the mapping required to transform world coordinates to normalized device coordinates (procedure TRY_TO_DEFINE_MAPPING).

Errors: 4 (The set of selected view surfaces is empty)

301 (There already is an open segment)

2.11 CLOSE_TEMPORARY_SEGMENT;

This procedure closes the current open temporary segment, preventing output primitives from being sent to selected view surfaces.

Errors: 307 (There is no open temporary segment)

2.12 QUERY_TEMPORARY_SEGMENT (VAR OPEN : CONDITION);

This procedure sets the parameter OPEN to the temporary segment status of the SSOCS library.

Errors: none.

2.13 SET_WINDOW (XMIN, XMAX, YMIN, YMAX : REAL);

This procedure is used to define the viewing specification's window. The parameters specify a rectangle in the world coordinate system. This rectangle represents the window; its left and right sides are vertical, and its top and bottom are horizontal. After it has been defined, the window may be rotated about the origin of the world coordinate system (see 2.15, SET_ VIEWUP).

The window and the viewport (see 2.19, SET_VIEWPORT) define the transformation which will be used to map from world coordinates to normalized device coordinates. When enabled, clipping will be performed at the window boundary, so that portions of objects which lie outside the window will not be visible. The contents of the window will be displayed on the viewport, which is established on a view surface.

The default specification for the window is (0.0, 1.0, 0.0, 1.0), and window clipping is enabled.

SET_WINDOW first checks that no segment is open, and that the supplied parameters are valid. It these conditions are satisfied, DEFINE_WINDOW is called to actually assign the parameters to the window variable, and an attempt is made to define the mapping from world coordinates to normalized device coordinates (procedure TRY_TO_DEFINE_MAPPING); this can only be done if the viewport has previously been specified.

Errors:	6	(A segment is open)
	501	(XMIN is not less than XMAX, or YMIN is not less than YMAX)

2.14 QUERY_WINDOW (VAR XMIN, XMAX, YMIN, YMAX : REAL);

This procedure assigns the coordinates specifying the current window to the appropriate parameters.

Errors: none.

2.15 SET_VIEWUP (DX_UP, DY_UP : REAL);

This procedure is used to define the world coordinate system 'up' direction. DX_UP and DY_UP define a 'view up point'. The window (see 2.13, SET_WINDOW) is rotated about the origin of the world coordinate system so that its left and right sides run in the direction of a vector from the origin to the view up point.

The default for (DX_UP, DY_UP) is (0.0, 1.0), so that the world coordinate Y axis is the view up direction.

SET_VIEWUP uses the specified parameters to calculate the angle of rotation of the window about the world coordinate origin, provided that no segment is open, and that the parameters are valid.

Errors:	6	(A segment is open)
	502	(DX_UP and DY_UP are both zero; no view up direction can be established)

2.16 QUERY_VIEWUP (VAR DX_UP, DY_UP : REAL);

This procedure assigns the coordinates specifying the view up point to the parameters DX_UP and DY_UP.

Errors: none.

2.17 SET_NDC_SPACE (WIDTH, HEIGHT : REAL);

This procedure defines the size of the two-dimensional normalized device coordinate space (NDC space) which can be addressed on the surfaces of display devices used by an application program. Viewports will be specified within this area (see 2.19, SET_VIEWPORT).

Both parameters must be in the range from 0.0 to 1.0, and at least one parameter must have a value of 1.0. Horizontally, normalized device coordinates range from 0.0 to WIDTH, and vertically from 0.0 to HEIGHT. The rectangle defined by the parameters is mapped to the viewable area of any display device so that the entire rectangle is visible.

The default normalized device coordinate specification is WIDTH = 1.0 and HEIGTH = 1.0.

SET_NDC_SPACE may only be invoked once for each initialization of the SSOCS library, and the normalized device coordinate space it establishes applies to all view surfaces that might be used by an application program. If SET_NDC_SPACE has not been called previously, the default values have not been assigned, and the specified parameters are valid, SET_NDC_SPACE calls DEFINE_NDC_SPACE to assign the parameters to the environment variable, and then:

1. creates the viewport to default specifications, if

the viewport has not been assigned by the application program, and

2. determines the mapping required to transform world coordinates to normalized device coordinates (procedure TRY_TO_DEFINE_MAPPING).

Three SSOCS procedures, CREATE_TEMPORARY_SEGMENT, SET_VIEWPORT, AND QUERY_VIEWPORT, require that the NDC space is established before they complete execution. If, prior to their invocation, SET_NDC_SPACE has not been used, these procedures will implicitly establish NDC space to the default specification.

Errors:	503	(SET_NDC_SPACE has already been invoked since the SSOCS library was last initialized)
	504	(The invocation of SET_NDC_SPACE is too late -- the default NDC space has been established)
	505	(A parameter is not in the range of 0.0 to 1.0)
	506	(Neither WIDTH nor HEIGHT has a value of 1.0)
	507	(WIDTH or HEIGHT is equal to zero)

2.18 QUERY_NDC_SPACE (VAR WIDTH, HEIGHT : REAL);

This procedure assigns the values specifying the normalized device coordinate space to the appropriate parameters.

Errors: none.

2.19 SET_VIEWPORT (XMIN, XMAX, YMIN, YMAX : REAL);

This procedure is used to define the viewing specification's viewport. The parameters specify a rectangle in normalized device coordinates. This rectangle represents the viewport; its sides are vertical, and top and bottom horizontal. The viewport establishes an area on a view surface, within which the contents of the window (see 2.13, SET_WINDOW) will be displayed.

The viewport cannot exceed the bound of normalized device coordinate space. The default viewport specification is the entire normalized device coordinate space.

Provided that there is no currently open segment (in which case the procedure terminates) SET_VIEWPORT calls CHECK_NDC_SPACE_STATE to determine the currently defined normalized device coordinate space. It then ensures that the supplied parameters lie within this space, and if so, calls DEFINE_VIEWPORT to assign the parameters to the viewport variable. SET_VIEWPORT then calls TRY_TO_DEFINE_MAPPING to determine the mapping required to transform world coordinates to normalized device coordinates.

Errors: 6 (A segment is open)

501 (XMIN is not less than XMAX, or YMIN is not less than YMAX)

508 (One or more of the viewport corners is outside the normalized device coordinate space)

2.20 QUERY_VIEWPORT (VAR XMIN, XMAX, YMIN, YMAX : REAL);

This procedure assigns the values specifying the viewport to the appropriate parameters.

Since it is possible for an application program to invoke this procedure without invoking SET_VIEWPORT, a call is made to CHECK_NDC_SPACE_STATE to ensure that, at least, normalized device coordinate space has been created to default specifications, and the default viewport has been defined. Once this has been completed, WHAT_IS_VIEWPORT is called to assign the viewport dimensioons to the parameters.

Errors: none.

2.21 MAP_NDC_TO_WORLD (NDC_X, NDC_Y : REAL; VAR X, Y : REAL);

This procedure calculates the world coordinates X and Y corresponding to the normalized device coordinates contained in NDC_X and NDC_Y.

The procedure calls WHAT_IS_VIEWPORT to determine the size of the current viewport, and ensures that the normalized device coordinates specified lie within this viewport. If this is so, TRANS_NDC_TO_WORLD is called to actually transform the coordinates, using the mapping established by the procedure TRY_TO_DEFINE_MAPPING. If the window is rotated, the reverse rotation is then applied to the coordinates (procedure ROTATE_POINT).

Errors: 510 (The specified normalized device coordinate position is outside the current viewport)

2.22 MAP_WORLD_TO_NDC (X, Y : REAL; VAR NDC_X, NDC_Y : REAL);

This procedure calculates the normalized device coordinates NDC_X and NDC_Y corresponding to the world coordinates contained in X and Y.

If clipping is enabled, the procedure calls WHAT_IS_WINDOW to determine the size of the current window, and ensure that the world coordinates specified lie within this window. If this is so, the following actions are performed:

1. if the window is rotated about the world coordinate origin, ROTATE_POINT is called to rotate the coordinates accordingly;
2. TRANS_WORLD_TO_NDC is called to actually transform the coordinates, using the mapping established by the

procedure TRY_ TO_DEFINE_MAPPING.

Errors: 512 (The specified world coordinate position is outside the current window and clipping is enabled)

2.23 SET_CLIPPING (ON_OFF : SWITCH);

This procedure is used to enable or disable clipping against the window in the viewing plane, according to the value of the parameter ON_OFF. When clipping is turned off, objects described in the world coordinate system are not checked for possible window clipping. If a line or a portion of a line appears outside the normalized device coordinate space the appearance of the line is device dependent. When clipping is enabled, objects described in the world coordinate system are clipped, if necessary, to the window.

The default window clipping mode is on.

Errors: 6 (A segment is open)

2.24 QUERY-CLIPPING (VAR MODE : SWITCH);

This procedure assigns the current clipping mode (either "on" or "off") to the parameter MODE.

Errors: none.

2.25 INITIALIZE_CORE;

This procedure must be the first SSOCS routine called by an application program. It guarantees that the SSOCS library is in a predefined state with the default settings of all parameters established.

INITIALIZE_CORE calls RESET_VALUES, which sets all SSOCS library parameters to their default state (showing, for example, that no segments are open, no errors have been detected). It then calls:

1. SET_WINDOW, to create the viewing operation window to the default specification;
2. SET_CLIPPING, to turn clipping on;
3. SET_VIEWUP, specifying a view up point at coordinates (0,1), showing no rotation of the window;
4. MOVE_ABS, to set the current pen position to the origin of the world coordinate system.

Errors: 701 (The SSOCS library has already been initialized)

2.26 TERMINATE_CORE;

This procedure is used to terminate use of the SSOCS library. It closes an open segment, terminates access to any initialized view surface (procedure MAKE_DEVICE), and releases any other resources used by the library.

After the SSOCS library has been terminated it may be reinitialized with the INITIALIZE_CORE procedure.

Errors: none.

2.27 INIT_VIEW_SURFACE (SURFACE_NAME : DEVICES);

This procedure is used to gain access to the view surface SURFACE_NAME, and to prepare that surface for graphics output. A device must be initialized with this procedure before it can be selected for output (see 2.29, SEL_VIEW_SURFACE).

The procedure adds the name SURFACE_NAME to the set of initialized surfaces, maintained in the Pascal set INIT_SURFACES. Procedure MAKE_DEVICE is than called to gain access to the surface, and to prepare it for graphics output.

Errors: 705 (The specified view surface is already initialized)

2.28 TERM_VIEW_SURFACE (SURFACE_NAME : DEVICES);

This procedure is used to terminate an application program's access to the view surface SURFACE_NAME, which will also be cleared.

SURFACE_NAME is removed from INIT_SURFACES, and

MAKE_DEVICE is called to clear the surface, and terminate access to it.

Errors: 708 (The specified view surface is not initialized)

2.29 SEL_VIEW_SURFACE (SURFACE_NAME : DEVICES);

This procedure adds the view surface SURFACE_NAME to the set of selected view surfaces SELECTED_SURFACES. Any images resulting from graphics output primitive function invocations will appear only on the view surfaces contained in the set of selected view surfaces. Also, when NEW_FRAME is called (see 2.31) a new-frame action will occur only on those view surfaces in this set.

The set of selected view surfaces is initially empty.

Errors:	6	(A segment is open)
	708	(The specified view surface is not initialized)
	709	(The specified view surface is already selected)

2.30 DESEL_VIEW_SURFACE (SURFACE_NAME : DEVICES);

This procedure removes the view surface SURFACE_NAME from SELECTED_SURFACES. Any subsequent graphics output, or new-frame actions, will not affect SURFACE-NAME unless it is reselected with a SEL_VIEW_SURFACE procedure invocation.

Errors:	6	(A segment is open)
	711	(The specified view surface is not selected)

2.31 NEW_FRAME;

This procedure causes a new-frame action to take place on each view surface that has been selected.

A new-frame action results in the elimination from view of all output primitives currently being displayed. For a terminal device, the display is cleared, while for a hardcopy device the output medium is advanced.

For each selected surface, NEW_FRAME calls MAKE_DEVICE to clear that surface.

Errors:	4	(The set of selected view surfaces is currently empty)

2.32 REPORT_MOST_RECENT_ERROR (VAR LATEST : ERRORDATA);

This procedure copies the report for the most recently detected error into LATEST, and then discards the original report. If no error has been detected since the SSOCS library was initialized, or since the last call to REPORT_LATEST_ERROR, a null error report will be returned (an error identifier and a severity code of zero).

Errors: none.

3. Error Messages and Severity Codes

This section shows the error messages, and their corresponding identification numbers, that may be generated by the SSOCS library:

2:	N is less than or equal to zero
4:	The set of selected view surfaces is empty
6:	A segment is open
201:	There is no open segment
301:	There already is an open segment
307:	There is no open temporary segment
501:	XMIN is not less than XMAX, or YMIN is not less than YMAX
502:	DX_UP and DY_UP are both zero; no view up direction can be established
503:	SET_NDC_SPACE has already been invoked since the SSOCS library was last initialized
504:	The invocation of SET_NDC_SPACE is too late -- the default NDC space has been established
505:	A parameter is not in the range of 0.0 to 1.0
506:	Neither WIDTH nor HEIGHT has a value of 1.0

507:	WIDTH or HEIGHT is equal to zero
508:	One or more of the viewport corners is outside the normalized device coordinate space
510:	The specified normalized device coordinate position is outside the current viewport
512:	The specified world coordinate position is outside the current window and clipping is enabled
701:	The SSOCS library has already been initialized
705:	The specified view surface is already initialized
708:	The specified view surface is not initialized
709:	The specified view surface is already selected
711:	The specified view surface is not selected

The severity codes that are used by the SSOCS library to notify the application program of the seriousness of an error are:

5: indicates an invalid parameter value or values; the procedure that has been invoked will return without any change to the state of the SSOCS library (for example, a parameter to SET_NDC_SPACE may be less than 0.0);

6: indicates that a routine has been called which is invalid with the present state of the SSOCS library (for example, an output primitive function is invoked when there is no open segment).

APPENDIX III

SOURCE CODE LISTING OF THE SSOCS LIBRARY

```
program ssocs_demo;

(*

        THIS IS THE PIXEL 100/AP (UNIX) VERSION OF THE SSOCS LIBRARY

        OWEN PLOWMAN,
        ISRAM

        APRIL, 1983


        Device drivers for:

                1.  VT125        (attached to serial 1)
                2.  GIGI         (    ----      serial 2)


        This version of SSOCS is based on the material found in
        "Status Report of The Graphics Standards Planning Committee",
        Computer Graphics, Volume 13, Number 3, August 1979.

 *)



  (*  CONSTANT DECLARATION SECTION
   *)


  const

    esc                 = '\1b';
    erroractual         = '/dev/console';  (*  error log device        *)
    maxsurfaces         = 4;               (*  surfaces available      *)

    ndcwidth            = 1.0;             (*  default width           *)
    ndcheight           = 1.0;             (*  default height          *)

    (*  constants for the devices

        VT125
     *)

    vt125actual         = '/dev/serial0';  (*  physical device for
                                                  the VT125             *)
```

```
vt125max            = 479;               (*  width of VT125 screen    *)
vt125home           = '[?61';            (*  home sequence for cursor*)
vt125alphaclear     = '[2j';             (*  clear alpha screen       *)
vt125graphicclear = 's(e)';              (*  clear graphics screen    *)
vt125gron           = 'Pp';              (*  enter graphics mode      *)
vt125groff          = '\\';              (*  exit graphics mode       *)
vt125setorigin      = 's(a[Ø,479][767,Ø])'; (*  origin to (Ø,Ø)       *)
vt125marker         = 't(s[8,14])';      (*  define marker size       *)

(*
   GIGI
 *)

gigiactual          = '/dev/serial2';   (*  physical device for
                                             the GIGI*)
gigimax             = 479;               (*  width of GIGI screen     *)
gigigraphicclear    = 's(e)';            (*  home and clear           *)
gigigron            = 'Pp';              (*  enter graphics mode      *)
gigigroff           = '\\';              (*  exit graphics mode       *)
gigisetorigin       = 's(a[Ø,479][767,Ø])'; (*  set origin to (Ø,Ø)*)
gigimarker          = 't(s[8,14])';      (*  define marker size       *)

(*
   VERSATEC  — not implemenented on this version
 *)

vstecmax            = 1Ø.555;            (*  width of versatec        *)
vstecletter         = Ø.21;              (*  size of characters       *)
vstecmarkoffset     = Ø.Ø7;              (*  to centre marker symbol *)

marksymbol          = '''.''';           (*  marker symbol             *)

maxpoints           = 8Ø;                (*  max. points for polyline*)
```

```
(*  TYPE DECLARATION SECTION
 *)

type

  switch = (on, off);

  operations = (up, down);        (*  interpreted as "operating state"  *)

  condition = (ouvert, ferme);    (*  since 'open' and
                                      'close' are reserved  *)

  buffer = string [5Ø];           (*  PIXEL-dependent type  *)
  coordrange = 1 .. maxpoints;
  coordseq = array [coordrange] of real;

  devices = (vt125, versatec, gigi, prism);
  surfaces = set of devices;

  vstecpens = Ø .. 9;   (*  pens available on the Versatec  *)

  surrange = 1 .. maxsurfaces;
  surarray = array [surrange] of devices;

  (*  The following are possible instructions to the command pathway
      to the device drivers
   *)
  instructions = (access, startup, clear, finishoff);

  radian = real;


  direction = (x, y);
  point = array [direction] of real;

  pair = (first, second);

  point_pair = record
                 p1,
                 p2 : point
               end;

  (*  This record holds information about the window in the graphical
      world (the application program's world)
   *)
  world_data = record
```

```
                fenster     : point_pair;  (*  window dimensions  *)
                clipping    : switch;      (*  clipping on/off  *)
                rotation    : radian;      (*  angle of rotation  *)
                viewuppoint : point        (*  current view up point  *)
              end;

   coordtrans = record                (*  for transforming points  *)
                 scale,               (*  from world to ndc coords *)
                 offset : real
               end;
   transformdata = array [direction] of coordtrans;

   (*  This record holds all of the information for the graphical
       world (the application program's world)
    *)
   environment = record
                   window    : world_data;
                   ndcspace  : point;
                   viewport  : point_pair;
                   mapping   : transformdata
                 end;

   (*  Data structures for describing primitives
    *)
   line_stuff = record
                ends  : point_pair;
                place : switch      (*  on or off the screen?  *)
              end;
   mark_stuff = record
                markposition  : point;
                place         : switch
              end;
   validescapes = (prompt, circle);
   escapestuff = record
                   viewsurface : devices;
                   case escapefunction : validescapes of
                     prompt :
                       (ndcx,
                        ndcy          : real;
                        promptmessage : buffer;);
                     circle :
                       (centre : point;
                        radius : real)
                 end;
   thingstodo = (line, marker, charseq, escapade);
   primitive = record
                 case action : thingstodo of
                   marker :
                         (currentmarker : mark_stuff);
                       line :
                         (currentline   : line_stuff);
                       escapade :
                         (currentescape : escapestuff)
                   end;


     (*  error reporting information
      *)
     error_info = (identifier, code);
     error_data = array [error_info] of integer;


     (*  The record defines here hold all of the information controlling
         the SSOCS system
      *)
     status_data = record
                       corestate,                        (*  system initialized?  *)
                       wndwstate,                        (*  window up?           *)
                       vportstate,                       (*  viewport up?         *)
                       ndcstate,                         (*  ndc space up?        *)
                       ndcdefstate     : operations;(*  ndc space set default*)
                       segstate        : condition; (*  segment open?*)
                       errorprocessing : switch;    (*  error handling on?   *)
                       reportgone      : boolean    (*  error report flushed?*)
                    end;
```

```
(*  VARIABLE DECLARATION SECTION
 *)


var

  cp  : point;                      (*  the current position  *)

  controls    : status_data;        (*  the state of SSOCS    *)

  the_world : environment;          (*  the graphical world   *)

  init_surfaces,                    (*  surfaces initialized  *)
  selected_surfaces : surfaces;     (*  surfaces selected     *)

  esccodes : set of validescapes;
  primcodes : set of thingstodo;

  error_report : error_data;

  stderr,
  gigifile,
  vt125file,
  prismfile : text;

  answer : char;

  rangle : radian;
  dpoint : point;
```

```
(*  SSOCS CONTROL VARIABLE PACKAGE MODULE
 *)


  procedure core_is (    state      : operations;
                     var controller : status_data);

  (*  This routine assigns the value specified by the parameter "state"
      to the SSOCS control variable ("state" specifies that the library
      is "up" or "down")
   *)

    begin
      with controller do
        corestate := state
    end;  (*  core_is  *)


  function core_is_down (controller : status_data) : boolean;

  (*  This routine returns the value true if the graphics system state is
      "down" (i.e. not initialized)
   *)

    begin
      with controller do
        core_is_down := corestate = down
    end;  (*  core_is_down  *)


  procedure a_segment_is (    state      : condition;
                          var controller : status_data);

  (*  This routine assigns the value specified by the parameter "state"
      to the segstate control variable, and establishes whether a segment
      is open or closed
   *)

    begin
      with controller do
        segstate := state
    end;  (*  setsegmentstate  *)


  function segment_is_open (controller : status_data) : boolean;

  (*  This routine returns the value true if there is currently a segment
    open
  *)

   begin
     with controller do
       segment_is_open := segstate = ouvert
   end;  (*  segment_is_open  *)


 procedure set_error_processing (    mode       : switch;
                                  var controller : status_data);

 (*  This routine assigns the value specified by the parameter "mode"
     to the errorprocessing control variable ("mode" specifies that
     error-processing is currently either "on" or "off"
  *)

   begin
     with controller do
       errorprocessing := mode
   end;  (*  set_error_processing  *)


 function handling_error (controller : status_data) : boolean;

 (*  This routine returns the value true if an error is currently being
     processed
  *)

   begin
     with controller do
       handling_error := errorprocessing = on
   end;  (*  handling_error  *)


 procedure check_call_is_valid (controller : status_data);

 (*  This routine is used to determine the validity of invocations of
     SSOCS routine by application programs.  No routine should be
     called while error-processing is going on.  If such a situation
     does arise, the SSOCS system will abort.
  *)

   begin
     if handling_error (controller) then
       halt
   end;  (*  check_call_is_valid  *)
```

```
procedure set_report_flushed (     flushed    : boolean;
                              var controller : status_data);

(*  This routine assigns the value specified by the parameter "flushed"
    to the reportflushed control variable ("flushed" specifies that an
    error report has (true) or has not (false) been flushed from the
    system
 *)

  begin
    with controller do
      reportgone := flushed
  end;  (*  set_report_flushed  *)


function report_is_flushed (controller : status_data) : boolean;

(*  This routine returns the value true if an error report has been
    flushed from the system
 *)

  begin
    with controller do
      report_is_flushed := reportgone
  end;  (*  report_is_flushed  *)


procedure the_window_is (     state      : operations;
                         var controller : status_data);

(*  This routine assigns the value specified by the parameter "state"
    to the windowstate control variable ("state" specifies that the
    window is currently either "up" or "down")
 *)

  begin
    with controller do
      wndwstate := state
  end;  (*  aswindowup  *)


function window_is_up (controller : status_data) : boolean;

(*  This routine returns the value true if the window is currently "up"
 *)

  begin
```

```
    with controller do
      window_is_up := wndwstate = up
  end;  (*  window_is_up  *)


procedure the_viewport_is (     state       : operations;
                            var controller : status_data);

(*  This routine assigns the value specified by the parameter "state"
    to the viewportstate control variable ("state" specifies that the
    viewport is currently either "up" or "down")
 *)

  begin
    with controller do
      vportstate := state
  end;  (*  the_viewport_is  *)


function viewport_is_up (controller : status_data) : boolean;

(*  This routine returns the value true if the viewport is currently "up"
 *)

  begin
    with controller do
      viewport_is_up := vportstate = up
  end;  (*  viewport_is_up  *)


procedure ndc_is (     state       : operations;
                   var controller : status_data);

(*  This routine assigns the value specified by the parameter "state"
    to the ndcspacestate control variable ("state" specifies that the
    ndc space is currently either "up" or "down")
 *)

  begin
    with controller do
      ndcstate := state
  end;  (*  ndc_is  *)


function ndc_is_up (controller : status_data) : boolean;

(*  This routine returns the value true is the ndc space as been
    specified (is "up")
```

```
   *)

    begin
      with controller do
        ndc_is_up := ndcstate = up
    end;  (*  ndc_is_up  *)


  procedure set_ndc_default_state (     state        : operations;
                                   var controller : status_data);

  (*  This routine assigns the value specified by the parameter "state"
      to the ndcspacedefaultstate control variable ["state" specifies
      that the ndc space default is currently either "up" or "down" (i.e.
      that the ndc space was or was not set to the default value)]
   *)

    begin
      with controller do
        ndcdefstate := state
    end;  (*  set_ndc_default_state  *)


  function ndc_default_was_set (controller : status_data) : boolean;

  (*  This routine returns the value true is the ndc space was set to the
      default values (i.e. if the default control variable for ndc space
      is "up")
   *)

    begin
      with controller do
        ndc_default_was_set := ndcdefstate = up
    end;  (*  ndc_default_was_set  *)
```

```
(*  ERROR PROCESSING ROUTINES

 The philosophy of error reporting and recovery supported by SSOCS is
 that all errors detected will be reported to the application program.

 Whenever an error is detected, SSOCS creates an error report, and in-
 vokes the function error_handler.  This routine takes a single
 aggregate parameter 'error_report', which is an ordered pair consisting
 of an error identifier and a severity code.  Both the error identifier
 and the severity code are unsigned integers.  The purpose of the severity
 code is to inform the application program of the seriousness of the
 particular error.

 A default error_handler is provided as part of SSOCS.  This function
 simply invokes the log_error routine.  log_error logs each error on
 an error log device, which is normally a file, and returns.  The
 default error_handler routine then returns control to SSOCS.  SSOCS
 always attempts to recover and continue, or at least to exit gracefully,
 if control can be returned.  For at least some errors, however, the
 graphical result cannot be guaranteed.

 For more sophisticated error-handling, an application program could
 provide its own error_handler by modifying the SSOCS source code.
 The only restriction on this application-provided function is that it
 not invoke any SSOCS function except log_error  (this is checked as
 each routine is called by the procedure check_call_is_valid).  If the
 application-supplied error_handler does invoke a SSOCS function, SSOCS
 will abort.

 The most recent error report is maintained by SSOCS for interrogation
 with the report_latest_error function.

 A SSOCS function invocation generates at most one error report.  The
 order in which SSOCS detects the various conditions is implementation-
 dependent (usually the error conditions are detected and reported in
 the order in which they appear in the GSPC Core System report).

 *)


 procedure process_error (errorid, severity : integer) ;

   forward;


 procedure build_error_report (     errorid,
                                    severity : integer;
                                    var report    : error_data);

  (*  This routine takes the values of errorid and severity, and creates
      a single aggregate variable containing both values
   *)

  begin
    report [identifier] := errorid;
    report [code] := severity
  end;  (*  build_error_report  *)


  procedure log_error (report : error_data);

  (*  This procedure can be used by the application program only as
      part of its error handling function.  It performs the logging
      that is accomplished by the default error handler.
   *)

  begin
    if handling_error (controls) then
      begin
        writeln (stderr, ' error number ', report [identifier] : 4,
                         ' has occurred');
        writeln (stderr, ' severity code = ', report [code] : 4)
      end
    else
      process_error (719, 6)
  end;  (*  log_error  *)


  procedure error_handler (error_report : error_data);

    begin
      log_error (error_report)
    end;  (*  error_handler  *)


  procedure process_error ;

  (*  This routine builds the error report for the current error, and
      invokes the error handling routine
   *)

  begin
    set_report_flushed (false, controls);
    build_error_report (errorid, severity, error_report);
    (*
         ... the current error report is passed to the errorhandling
         routine.  The errorprocessing flag is set to "on", to ensure
         that no other system routines are called while error
         processing is going on.  The error processing flag is turned off
         when the errorhandling routine returns
     *)
    set_error_processing (on, controls);
    error_handler (error_report);
    set_error_processing (off, controls)
  end;  (*  process_error  *)


  procedure report_latest_error (var latest : error_data);

  (*  This procedure copies the error report for the most recently
      detected error into latest, and flushes the report from the SSOCS
      system.  If no error has occurred since SSOCS was initialized,
      or since report_latest_error was last invoked, the procedure
      returns a null error report (an error identifier and a severity
      code of zero).
   *)

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
        (*
            ... if there is a "latest error", it is copied into the
            variable latest.  Otherwise a null error report is returned.
            The control variable for the flushing of reports is set to
            true, indicating that a report has been flushed
         *)
      begin
        if report_is_flushed (controls) then
          build_error_report (Ø, Ø, latest)
        else
          build_error_report (error_report [identifier],
                              error_report [code], latest);
        set_report_flushed (true, controls)
      end
  end;  (*  report_latest_error  *)
```

```
(*  GRAPHICS ENVIRONMENT PACKAGE  *)


(*  encapsulation routines for a 'point'  *)

  procedure set_coordinates (    xcoord,
                                 ycoord    : real;
                             var thepoint  : point);

  (*  constructor  *)

    begin
      thepoint [x] := xcoord;
      thepoint [y] := ycoord
    end;  (*  set_coordinates  *)


  function get_x_coord (anypoint : point) : real;

  (*  selector function for x coordinate  *)

    begin
      get_x_coord := anypoint [x]
    end;  (*  get_x_coordinate  *)


  function get_y_coord (anypoint : point) : real;

  (*  selector function for y coordinate  *)

    begin
      get_y_coord := anypoint [y]
    end;  (*  get_y_coordinate  *)


  function points_equal (p1, p2 : point) : boolean;

  (*  function to test two points for equality  *)

    begin
      points_equal := (get_x_coord (p1) = get_x_coord (p2)) and
                      (get_y_coord (p1) = get_y_coord (p2))
    end;  (*  points_equal  *)
```

```
(*  encapsulation routines for a 'point_pair'  *)


procedure set_a_point (     thepoint : point;
                            which    : pair;
                        var anypair  : point_pair);

  begin
    with anypair do
      if which = first then
        p1 := thepoint
      else
        p2 := thepoint
  end;  (*  set_a_point  *)


procedure get_a_point (var thepoint : point;
                           which    : pair;
                           anypair  : point_pair);

  begin
    with anypair do
      if which = first then
        thepoint := p1
      else
        thepoint := p2
  end;  (*  get_a_point  *)


procedure define_rectangle (     xmin, xmax,
                                 ymin, ymax : real;
                             var anybox     : point_pair);

  var
    dummy : point;

  begin
    set_coordinates (xmin, ymin, dummy);
    set_a_point (dummy, first, anybox);
    set_coordinates (xmax, ymax, dummy);
    set_a_point (dummy, second, anybox)
  end;  (*  define_rectangle  *)


procedure what_is_rectangle (var xmin, xmax,
                                 ymin, ymax : real;
                                 anybox     : point_pair);
    var
      dummy : point;

    begin
      get_a_point (dummy, first, anybox);
      xmin := get_x_coord (dummy);
      ymin := get_y_coord (dummy);
      get_a_point (dummy, second, anybox);
      xmax := get_x_coord (dummy);
      ymax := get_y_coord (dummy)
    end;  (*  what_is_rectangle  *)


  (*  encapsulation routines for the graphical environment  *)


  procedure define_window (     xmin, xmax,
                                ymin, ymax : real;
                            var anyworld   : environment);

    begin
      with anyworld do
        with window do
          define_rectangle (xmin, xmax, ymin, ymax, fenster)
    end;  (*  define_window  *)


  procedure what_is_window (var xmin, xmax,
                                ymin, ymax : real;
                                anyworld   : environment);

    begin
      with anyworld do
        with window do
          what_is_rectangle (xmin, xmax, ymin, ymax, fenster)
    end;  (*  what_is_window  *)


  procedure clipping_is (     mode      : switch;
                          var anyworld : environment);

    begin
      with anyworld do
        with window do
          clipping := mode
    end;  (*  clipping_is  *)
```

```
function clip_switch (anyworld : environment) : switch;

  begin
    with anyworld do
      with window do
        clip_switch := clipping
  end;  (*  clip_switch  *)


procedure angle_is (    angle    : radian;
                    var anyworld : environment);

  begin
    with anyworld do
      with window do
        rotation := angle
  end;  (*  angle_is *)


function get_angle (anyworld : environment) : radian;

  begin
    with anyworld do
      with window do
        get_angle := rotation
  end;  (*  get_angle  *)


procedure define_vu_point (    xcoord,
                               ycoord   : real;
                           var anyworld : environment);

  begin
    with anyworld do
      with window do
        set_coordinates (xcoord, ycoord, viewuppoint)
  end;  (*  define_vu_point  *)


procedure what_is_vu_point (var xcoord,
                                ycoord   : real;
                                anyworld : environment);

  begin
    with anyworld do
      with window do
```

```
        begin
          xcoord := get_x_coord (viewuppoint);
          ycoord := get_y_coord (viewuppoint)
        end
  end;  (*  what_is_vu_point  *)


procedure scale_is (    scalevalue    : real;
                        forcoordinate : direction;
                    var anyworld      : environment);

  begin
    with anyworld do
      with mapping [forcoordinate] do
        scale := scalevalue
  end;  (*  scale_is  *)


procedure offset_is (    offsetvalue   : real;
                         forcoordinate : direction;
                     var anyworld      : environment);

  begin
    with anyworld do
      with mapping [forcoordinate] do
        offset := offsetvalue
  end;  (*  offset_is  *)


function get_scale (forcoordinate : direction;
                    anyworld      : environment) : real;

(*  This routine finds the scaling factor for the specified world
    to ndc space mapping, for the specified direction  *)

  begin
    with anyworld do
      with mapping [forcoordinate] do
        get_scale := scale
  end;  (*  get_scale  *)


function get_offset (forcoordinate : direction;
                     anyworld      : environment) : real;

(*  This routine finds the offset factor for the specified world
    to ndc space mapping, for the specified direction  *)
    begin
      with anyworld do
        with mapping [forcoordinate] do
          get_offset := offset
    end;  (* get_offset  *)


  procedure define_viewport (    xmin, xmax,
                                 ymin, ymax : real;
                             var anyworld   : environment);

    begin
      with anyworld do
        define_rectangle (xmin, xmax, ymin, ymax, viewport)
    end;  (* define_viewport  *)


  procedure what_is_viewport (var xmin, xmax,
                                  ymin, ymax : real;
                                  anyworld   : environment);

    begin
      with anyworld do
        what_is_rectangle (xmin, xmax, ymin, ymax, viewport)
    end;  (* what_is_viewport  *)


  procedure trans_ndc_to_world (var p : point);

  (*  This routine performs a simple transformation of a point in
      ndc space to world space  *)

    var
      xnew,
      ynew : real;

    begin
      xnew := (get_x_coord (p) - get_offset (x, the_world)) /
                get_scale (x, the_world);
      ynew := (get_y_coord (p) - get_offset (y, the_world)) /
                get_scale (y, the_world);
      set_coordinates (xnew, ynew, p)
    end;  (*  trans_ndc_to_world  *)


  procedure trans_world_to_ndc (var p : point);

  (*  This routine performs a simple transformation of a point in
      world space to ndc space  *)

    var
      xndc,
      yndc : real;

    begin
      xndc := (get_x_coord (p) * get_scale (x, the_world)) +
               get_offset (x, the_world);
      yndc := (get_y_coord (p) * get_scale (y, the_world)) +
               get_offset (y, the_world);
      set_coordinates (xndc, yndc, p)
    end;  (*  trans_world_to_ndc  *)


  procedure try_to_define_mapping (var anyworld : environment);

   (*
     This procedure attempts to set up the mapping used to transform
     coordinates from the world to the viewport.

     If both the window and the viewport have been established, then
     the mapping will also be established, based on the current bounds
     of the window and viewport.
    *)

    var
      vxr, vxl,
      vyt, vyb,
      wxr, wxl,
      wyt, wyb : real;

    begin
      if window_is_up   (controls) and
         viewport_is_up (controls) then
        begin
          (*
              ... both the window and the viewport have been established.
              Their boundary values are determined, and the mapping is
              then calculated
           *)
          what_is_window   (wxl, wxr, wyb, wyt, anyworld);
          what_is_viewport (vxl, vxr, vyb, vyt, anyworld);
          (*
              ... for x coordinates
           *)
          scale_is (((vxr - vxl) / (wxr - wxl)), x, anyworld);
          offset_is ((vxl - (wxl * get_scale (x, anyworld))), x, anyworld);
        (*
            ... for y coordinates
         *)
        scale_is (((vyt - vyb) / (wyt - wyb)), y, anyworld);
        offset_is ((vyb - (wyb * get_scale (y, anyworld))), y, anyworld)
      end
  end;  (*  try_to_define_mapping  *)


(*  point rotation  *)


procedure rotate_point (var thepoint : point;
                            theta    : radian);

  (*  This routine performs a rotation of a point by the specified
      angle  *)

  var
    x, y,
    xprime, yprime : real;

  begin
    x := get_x_coord (thepoint);
    y := get_y_coord (thepoint);
    xprime := (x * cos (theta)) + (y * sin (theta));
    yprime := ((-1) * x * sin (theta)) + (y * cos (theta));
    set_coordinates (xprime, yprime, thepoint)
  end;  (*  rotate_point  *)


procedure define_ndc_space (    width,
                                height    : real;
                            var anyworld : environment);

  begin
    with anyworld do
      set_coordinates (width, height, ndcspace)
  end;  (*  define_ndc_space  *)


procedure what_is_ndc_space (var width,
                                 height   : real;
                                 anyworld : environment);

  begin
    with anyworld do
    begin
      width  := get_x_coord (ndcspace);
      height := get_y_coord (ndcspace)
    end
end;  (*  what_is_ndc_space  *)
```

```
(*  PRIMITIVE PACKAGE  *)


  function sel_surface (thelist : escapestuff) : devices;

    begin
      with thelist do
        sel_surface := viewsurface
    end;  (*  sel_surface  *)


  procedure as_surface (    thesurface : devices;
                        var thelist    : escapestuff);

    begin
      with thelist do
        viewsurface := thesurface
    end;  (*  as_surface  *)


  function sel_function (thelist : escapestuff) : validescapes;

    begin
      with thelist do
        sel_function := escapefunction
    end;  (*  sel_function *)


  procedure as_function (    thefunction : validescapes;
                         var thelist     : escapestuff);

    begin
      with thelist do
        escapefunction := thefunction
    end;  (*  as_function *)


  procedure sel_prompt_start (var p       : point;
                                  thelist : escapestuff);

    begin
      with thelist do
        if sel_function (thelist) = prompt then
          set_coordinates (ndcx, ndcy, p)
        else
          process_error (1ØØ1, 6)
    end;  (*  sel_prompt_start  *)
```

```
procedure as_prompt_start (    x, y     : real;
                           var thelist : escapestuff);

  begin
    with thelist do
      if sel_function (thelist) = prompt then
        begin
          ndcx := x;
          ndcy := y
        end
      else
        process_error (1ØØ1, 6)
  end;  (*  as_prompt_start  *)


procedure sel_message (var themessage : buffer;
                           thelist    : escapestuff);

  begin
    with thelist do
      if sel_function (thelist) = prompt then
        themessage := promptmessage
      else
        process_error (1ØØ1, 6)
  end;  (*  sel_message  *)


procedure as_message (    themessage : buffer;
                      var thelist    : escapestuff);

  begin
    with thelist do
      if sel_function (thelist) = prompt then
        promptmessage := themessage
      else
        process_error (1ØØ1, 6)
  end;  (*  as_message  *)


procedure sel_centre (var p       : point;
                          thelist : escapestuff);

  begin
    with thelist do
      if sel_function (thelist) = circle then
        p := centre
      else
        process_error (1001, 6)
    end;  (*  sel_centre  *)


  procedure as_centre (     p        : point;
                       var thelist : escapestuff);

    begin
      with thelist do
        if sel_function (thelist) = circle then
          centre := p
        else
          process_error (1001, 6)
    end;  (*  as_centre  *)


  function sel_radius (thelist : escapestuff) : real;

    begin
      with thelist do
        if sel_function (thelist) = circle then
          sel_radius := radius
        else
          process_error (1001, 6)
    end;  (*  sel_radius  *)


  function get_function (instance : primitive) : validescapes;

    begin
      with instance do
        get_function := sel_function (currentescape)
    end;  (*  get_function  *)


  function get_surface (instance : primitive) : devices;

    begin
      with instance do
        get_surface := sel_surface (currentescape)
    end;  (*  get_surface  *)


  procedure get_message (var line     : buffer;
                             instance : primitive);

    begin
```

```
      with instance do
        sel_message (line, currentescape)
    end;  (*  get_message  *)


  function get_radius (instance : primitive) : real;

    begin
      with instance do
        get_radius := sel_radius (currentescape)
    end;  (*  get_radius  *)


  procedure set_action_to (    thisaction : thingstodo;
                           var instance   : primitive);

    begin
      with instance do
        action := thisaction;
    end;  (*  set_action_to  *)


  function get_the_action (instance : primitive) : thingstodo;

    begin
      with instance do
        get_the_action := action;
    end;  (*  get_the_action  *)


  procedure set_end_points (    beginning,
                                ending     : point;
                            var instance   : primitive);

    begin
      if get_the_action (instance) in primcodes then
        with instance do
          case get_the_action (instance) of
            line :
              with currentline do
                begin
                  set_a_point (beginning, first, ends);
                  set_a_point (ending, second, ends)
                end;
            marker, escapade :
              process_error (1ØØ1, 6)
          end
      else
```

```
        process_error (1000, 6)
      end;  (*  setlineends  *)


    procedure get_end_points (var beginning,
                                  ending    : point;
                                  instance  : primitive);

      begin
        if get_the_action (instance) in primcodes then
          with instance do
            case get_the_action (instance) of
              line :
                with currentline do
                  begin
                    get_a_point (beginning, first, ends);
                    get_a_point (ending, second, ends)
                  end;
              marker, escapade :
                process_error (1001, 6)
            end
        else
          process_error (1000, 6)
      end;  (*  get_end_points  *)


    procedure the_line_is (    onscreen : switch;
                           var instance : primitive);

      begin
        if get_the_action (instance) in primcodes then
          with instance do
            case get_the_action (instance) of
              line :
                with currentline do
                  place := onscreen;
              marker, escapade :
                process_error (1001, 6)
            end
        else
          process_error (1000, 6)
      end;  (*  the_line_is  *)


    function line_is_on (instance : primitive) : boolean;

      begin
        if get_the_action (instance) in primcodes then
        with instance do
          case get_the_action (instance) of
            line :
              with currentline do
                line_is_on := place = on;
            marker, escapade :
              process_error (1ØØ1, 6)
          end
      else
        process_error (1ØØØ, 6)
    end;  (*  line_is_on  *)


  procedure go_to_action_point (    position : point;
                                var instance : primitive);

    begin
      if get_the_action (instance) in primcodes then
        with instance do
          case get_the_action (instance) of
            marker :
              with currentmarker do
                markposition := position;
            line, escapade :
              process_error (1ØØ1, 6)
          end
      else
        process_error (1ØØØ, 6)
    end;  (*  go_to_action_point  *)


  procedure what_is_action_point (var thepoint  : point;
                                      instance : primitive);

    begin
      if get_the_action (instance) in primcodes then
        with instance do
          case get_the_action (instance) of
            marker :
              with currentmarker do
                thepoint := markposition;
            escapade :
              case sel_function (currentescape) of
                prompt :
                  sel_prompt_start (thepoint, currentescape);
                circle :
                  sel_centre (thepoint, currentescape)
              end;
          line :
            process_error (1001, 6)
        end
    else
      process_error (1000, 6)
  end;  (*  what_is_action_point  *)


procedure the_point_is (    onscreen : switch;
                        var instance : primitive);

  begin
    if get_the_action (instance) in primcodes then
      with instance do
        case get_the_action (instance) of
          marker :
            with currentmarker do
              place := onscreen;
          line, escapade :
            process_error (1001, 6)
        end
    else
      process_error (1000, 6)
  end;  (*  the_point_is  *)


function point_is_on (instance : primitive) : boolean;

  begin
    if get_the_action (instance) in primcodes then
      with instance do
        case get_the_action (instance) of
          marker :
            with currentmarker do
              point_is_on := place = on;
          line, escapade :
            process_error (1001, 6);
        end
    else
      process_error (1000, 6)
  end;  (*  point_is_on  *)


procedure set_parameters (    parmlist  : escapestuff;
                          var instance  : primitive);

  begin
    if get_the_action (instance) in primcodes then
```

```
      with instance do
        case get_the_action (instance) of
          escapade :
            currentescape := parmlist;
          line, marker :
            process_error (1ØØ1, 6)
        end
    else
      process_error (1ØØØ, 6)
  end;  (*  set_parameters  *)
```

```
(*  VIEWING OPERATIONS  *)


 procedure set_window (xmin, xmax, ymin, ymax : real);

  (*
    The parameters to set_window specify a rectangle (in world
    coordinates, which has left and right sides vertical, and top
    and bottom horizontal.  The rectangle is rotated about the origin
    of the world coordinate system so that the sides running from
    ymin to ymax run in the direction of the view-up vector.  This
    rotated rectangle defines the window.  In the default case,
    the view-up vector is in the positive y direction, so that no
    rotation has to be performed.

    The window and the viewport define the transformation used to
    map from world coordinates to normalized device coordinates.
    When enabled, clipping is performed at the window boundary.
    That is, portions of objects on or inside the window boundary are
    visible, while portions outside the boundary are not.  If window
    clipping is disabled, then the entire xy plane is mapped to the
    view surface in such a way that the window is mapped to the
    viewport.  The image appearing on the view surface is
    well-defined only for that portion of the xy plane which,
    when mapped to the view surface, falls within the range of normal-
    ized device coordinate space.  Any object whose image is partially
    or completely outside these bounds will be displayed in an imp-
    lementation dependent way.

    The default window specification is (0.0, 1.0, 0.0, 1.0), and window
    clipping is enabled.
  *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if segment_is_open (controls) then
        process_error (6, 6)
      else if ((xmin >= xmax) or
               (ymin >= ymax)) then
        process_error (501, 5)
      else
        begin
          (*
            ... the values specified for the boundaries of the window
          are valid, and so the window is defined.  The status of
          the window is set to "up", and an attempt is made to
          determine the world --> ndc-space mapping (this can only
          be done if the viewport has also been defined).
         *)
        define_window (xmin, xmax, ymin, ymax, the_world);
        the_window_is (up, controls);
        try_to_define_mapping (the_world);
      end
  end;  (*  set_window  *)


procedure query_window (var xmin, xmax, ymin, ymax : real);

(*
  The values specifying the current window are copied
  into the parameters.
 *)

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      what_is_window (xmin, xmax, ymin, ymax, the_world)
  end;  (*  query_window  *)


procedure set_clipping (mode : switch);

(*
  This procedure is used to enable or disable clipping
  against the window in the view plane.  When window clipping
  is "off", objects described to the SSOCS system are not
  checked for possible window clipping.  If a line, or a
  portion of a line, appears outside the normalized device
  coordinate space, the appearance of that line is undefined.
  When window clipping is set to "on", objects described to
  the SSOCS system are clipped, when necessary.

  The default window clipping mode is "on".
 *)

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else if segment_is_open (controls) then
```

```
      process_error (6, 6)
    else
      clipping_is (mode, the_world)
  end;  (*  set_clipping  *)


procedure query_clipping (var mode : switch);

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      mode := clip_switch (the_world)
  end;  (*  query_clipping  *)


procedure set_view_up (dxup, dyup : real);

(*
  This function specifies the world coordinate "up"
  direction, so that the world coordinate y-axis need not be
  upright on the view surface.  The synthetic camera is
  rotated about its line of sight so that the vector from
  the origin to (dxup, dyup) would appear to be vertical in
  the camera's viewfinder.

  The default for (dxup, dyup) is (Ø.Ø, 1.Ø), and so the world
  coordinate y-axis direction is the default view up direction.
*)


  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else if segment_is_open (controls) then
      process_error (6, 6)
    else if (dxup = Ø.Ø) and
            (dyup = Ø.Ø) then
      process_error (5Ø2, 5)
    else
      begin
        define_vu_point (dxup, dyup, the_world);
        if (dxup = Ø.Ø) and
           (dyup > Ø.Ø) then
          angle_is (Ø.Ø, the_world)
        else
```

```
          angle_is ((-1.Ø) * (arctan (dxup / dyup)), the_world)
      end
  end;  (*  set_view_up  *)


procedure query_vw_up (var dxup, dyup : real);

(*
  The coordinates for the view up point are copied into the
  specified parameters.
 *)

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      what_is_vu_point (dxup, dyup, the_world)
  end;  (*  query_vw_up  *)


procedure set_ndc_space (width, height : real);

(*
  This function defines the size of the two-dimensional
  normalized device coordinate space which can be addressed on
  the view surface of all the display devies used by the
  application program, and within which viewports will be
  specified.  Both parameters must be in the range Ø to 1, and
  at least one parameter must have a value of 1.

  Horizontally, normalized device coordinates range from Ø
  to width, and vertically, from Ø to height.  The rectangle
  so defined is mapped to the viewable area of any display
  device used by the application program, so that the entire
  rectangle is visible.  Only uniform scaling of the rectangle
  is allowed as part of the mapping, which will usually (but
  not necessarily) maximize the useable area of the display.

  The default normalized device coordinate specification is
  width = 1.Ø and height = 1.Ø.  set_ndc_space may only be
  used once for each initialization of the SSOCS system, and
  the ndc space it establishes applies to all view surfaces
  which might be used by the application program.

  Several SSOCS system routines require that ndc space is established
  before they complete execution.  If, prior to their invocation, the
  ndc space has not been established, these routines will implicitly
```

```
    establish ndc space using the default values.  The routines are:

        create_segment,
        set_viewport,
        query_viewport
    *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if ndc_is_up (controls) then
        process_error (503, 6)
      else if ndc_default_was_set (controls) then
        process_error (504, 6)
      else if (width < 0.0) or
              (width > 1.0) or
              (height < 0.0) or
              (height > 1.0) then
        process_error (505, 5)
      else if (width  <> 1.0)  and
              (height <> 1.0) then
        process_error (506, 5)
      else if (width  = 0.0) or
              (height = 0.0) then
        process_error (507, 5)
      else
        begin
          (*
            ... the input values are valid, so the ndc space is
            defined.  The status of the ndc space is set to "up".
           *)
          define_ndc_space (width, height, the_world);
          ndc_is (up, controls);
          (*
            ... if the viewport has not yet been specified, it
            is defined as being equal to the entire ndc space,
            and an attempt is made to establish the world -->
            ndc-space mapping(this can only be done if the
            window has also been specified).
           *)
          if not viewport_is_up (controls) then
            begin
              define_viewport (0.0, width, 0.0, height, the_world);
              the_viewport_is (up, controls);
              try_to_define_mapping (the_world);

            end
        end
    end;  (*  set_ndc_space  *)


  procedure query_ndc_space (var width, height : real);

  (*
    The current dimensions of the normalized device coordinate
    space is copied into the specified parameters.
   *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else
        what_is_ndc_space (width, height, the_world)
    end;  (*  query_ndc_space  *)


  procedure check_ndc_space_state ;

  (*
    This procedure is used by certain SSOCS system functions
    to determine whether the ndc space has been established.
    If either an application program or another of the SSOCS system
    functions has set the ndc space, the procedure exits.  Otherwise
    the ndc space is set to the default value (width = 1.0 and
    height = 1.0), and the control flag is set to show that this
    operation has been carried out.
   *)

    begin
      if ndc_is_up (controls) then
        (*  ndc space has been set; do nothing  *)
      else if ndc_default_was_set (controls) then
        (*  ndc space has been set by another system function;
            do nothing  *)
      else
        begin
          (*
              ... the ndc space has not yet been set, and so it is set to
              the default values.  The status of the ndc space is set to
              "up", and indicates that ndc space was set by default
           *)
          define_ndc_space (ndcwidth, ndcheight, the_world);
          set_ndc_default_state (up, controls);
          (*
            ... if the viewport has not yet been set, it is made equal to
            the default ndc space, and an attempt is made to establish the
            world ---> ndc space mapping (this can only be done if the
            window has also been specified
          *)
         if not viewport_is_up (controls) then
           begin
             define_viewport (Ø.Ø, ndcwidth, Ø.Ø, ndcheight, the_world);
             the_viewport_is (up, controls);
             try_to_define_mapping (the_world);
           end
       end
   end;  (*  check_ndc_space_state  *)


  procedure set_viewport (xmin, xmax, ymin, ymax : real);

  (*  The parameters give the extent, in two-dimensional normalized device
      coordinate space, of the current viewport.  The viewport's sides are
      vertical, and its top and bottom horizontal.  The viewport cannot
      exceed the boundaries of normalized device coordinate space.

      This viewport will be used for displaying all output primitives until a
      new viewport is specified.  The default viewport specification is the
      entire normalized device coordinate space, as specified by set_ndc_space,
      or the default (width = 1.Ø, height = 1.Ø), if this function has not
      been invoked.
  *)

    var
      width,
      height: real;

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if segment_is_open (controls) then
        process_error (6, 6)
      else if (xmin >= xmax) or
              (ymin >= ymax) then
        process_error (5Ø1, 5)
      else
        begin
          the_viewport_is (up, controls);
          check_ndc_space_state ;
          the_viewport_is (down, controls);
          query_ndc_space (width, height);
        if (xmin < Ø.Ø) or
           (xmax > width) or
           (ymin < Ø.Ø) or
           (ymax > height) then
          process_error (5Ø8, 5)
        else
          begin
            define_viewport (xmin, xmax, ymin, ymax, the_world);
            the_viewport_is (up, controls);
            try_to_define_mapping (the_world)
          end
      end
  end;  (*  set_viewport  *)


procedure query_viewport (var xmin, xmax, ymin, ymax : real);

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      begin
        check_ndc_space_state ;
        what_is_viewport (xmin, xmax, ymin, ymax, the_world)
      end
  end;  (*  query_viewport  *)


procedure map_ndc_to_world (     ndcx, ndcy : real;
                             var wx,   wy   : real);

 (*  The world coordinates corresponding to the position (ndcx, ndcy)
     are calculated, and written into the parameters x and y.  *)

  var
    xmin, xmax,
    ymin, ymax : real;
    dummy : point;

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      begin
        (*
            ... determine the size of the viewport and ensure that
```

```
              the coordinates to be transformed lie within it
           *)
          what_is_viewport (xmin, xmax, ymin, ymax, the_world);
          if (ndcx < xmin) or
             (ndcx > xmax) or
             (ndcy < ymin) or
             (ndcy > ymax) then
            process_error (510, 5)
          else
            begin
              (*
                  ... build a dummy point carrying the required
                  coordinates and transform it to world space
               *)
              set_coordinates (ndcx, ndcy, dummy);
              trans_ndc_to_world (dummy);
              (*
                  ... if the window is rotated, apply the required
                  (reverse) rotation to the dummy point
               *)
              if get_angle (the_world) <> 0.0 then
                rotate_point (dummy, (-1.0) * get_angle (the_world));
              (*
                  ... retrieve the now-transformed coordinates
               *)
              wx := get_x_coord (dummy);
              wy := get_y_coord (dummy)
            end
        end
    end;  (*  map_ndc_to_world  *)


  procedure map_world_to_ndc (     x, y          : real;
                               var ndcx, ndcy : real);

  (*
    ... the normalized device coordinates corresponding to
    the world position (x, y) are calculated, and written
    into ndcx and ndcy.
   *)

    var
      xmin, xmax,
      ymin, ymax : real;
      dummy : point;

    begin
      check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else
      begin
        (*
             ... determine the size of the window, and ensure that the
             coordinates to be transformed lie within it if clipping
             to the window boundary is enabled
         *)
        what_is_window (xmin, xmax, ymin, ymax, the_world);
        if ((x < xmin) or
            (x > xmax) or
            (y < ymin) or
            (y > ymax)) and
            (clip_switch (the_world) = on) then
          process_error (512, 5)
        else
          begin
            (*
                 ... build a dummy point carrying the specified
                 coordinates.
             *)
            set_coordinates (x, y, dummy);
            (*
                 ... if the window is rotated, rotate the point
                 accordingly, and then transform the coordinates
                 to ndc space.
             *)
            if get_angle (the_world) <> 0.0 then
              rotate_point (dummy, get_angle (the_world));
            trans_world_to_ndc (dummy);
            (*
                 ... retrieve the now-transformed coordinates
             *)
            ndcx := get_x_coord (dummy);
            ndcy := get_y_coord (dummy)
          end
        end
  end;  (*  map_world_to_ndc  *)
```

```
(*  DEVICE DRIVER ROUTINES  *)


(*  The following declarations are for the Cyber version only.

    External fortran routines for the versatec plotter


  procedure plot (x, y : real; code : integer);

    fortran;

  procedure letter (     n       : integer;
                         height,
                         angle,
                         xl,
                         yl      : real;
                     var bcd     : buffer);

    fortran;

  procedure newpen (penwidth : vstecpens);

    fortran;

  procedure arc (xa, ya, xb, yb, xc, yc, dev : real);

    fortran;

  ... end of Versatec external declarations  *)



  (*  device driver routines  *)


  procedure make_device (docode      : instructions;
                         thedevice : devices);

  (*  This is the command pathway for the SSOCS device drivers.
      Any action which is concerned with the state of a device
      is performed from this routine
   *)

    begin
      case thedevice of
    vt125 :
      case docode of
        access :
          rewrite (vt125file, vt125actual);
        startup :
          begin
            write (vt125file, esc, vt125home,
                                  esc, vt125alphaclear);
            write (vt125file, esc, vt125gron,
                                  esc, vt125graphicclear, vt125setorigin)
          end;
        clear :
          write (vt125file, esc, vt125graphicclear);
        finishoff :
          write (vt125file, esc, vt125groff)
      end;
    gigi :
      case docode of
        access :
          rewrite (gigifile, gigiactual);
        startup :
          write (gigifile, esc, gigigron,
                           gigigraphicclear, gigisetorigin);
        clear :
          write (gigifile, gigigraphicclear);
        finishoff :
          write (gigifile, esc, gigigroff)
      end;
    versatec :          (*  THIS DEVICE IS UNAVAILABLE  *)
      case docode of
        access :
          (*  no special action required  *);
        startup :
          (*  newpen (2)  *);
        clear :
          (*  plot (0.0, 0.0, 999)  *);
        finishoff :
          (*  no special action required  *)
      end;
    prism :
      (*
          ... not yet implemented
       *)
  end
end;  (*  make_device  *)
```

```
(* The following routine make up the graphical command pathway for
   the SSOCS system. Any graphical output is performed through these
   routines.  If a device cannot support an action (for example,
   low-quality text output), it must be simulated in the
   routine provided here
 *)


 procedure vt125line (p1, p2 : point);

   var
     x, y : integer;

   begin
     (*  ... find the coordinates of the starting position of the
         line and convert them to device coordinates  *)
     x := round (get_x_coord (p1) * vt125max);
     y := round (get_y_coord (p1) * vt125max);
     (*  ... set the current position, for the device, to the start
         of the line  *)
     write (vt125file, 'p[', x, ',', y, ']');
     (*  ... similarly, find the coordinates of the end point of the
         line, and convert them to device coordinates  *)
     x := round (get_x_coord (p2) * vt125max);
     y := round (get_y_coord (p2) * vt125max);
     (*  ... draw a vector on the device, from the current position
         to the end point of the line  *)
     write (vt125file, 'v[', x, ',', y, ']')
   end;  (*  vt125line  *)


 procedure vt125mark (markpoint : point);

   var
     x, y : integer;

   begin
     (*  ... find the coordinates of the marker symbol and convert
         them to device coordinates  *)
     x := (round (get_x_coord (markpoint) * vt125max)) - 4;
     y := (round (get_y_coord (markpoint) * vt125max)) + 12;
     (*  ... go to the point specified on the device screen  *)
     write (vt125file, 'p[', x, ',', y, ']');
     (*  ... write the marker symbol at the current point  *)
     write (vt125file, vt125marker, marksymbol)
   end;  (*  vt125mark  *)
```

```
procedure vt125text (atpoint : point;
                     message  : buffer);

  var
    x, y : integer;

  begin
    x := (round (get_x_coord (atpoint) * vt125max));
    y := (round (get_y_coord (atpoint) * vt125max));
    write (vt125file, 'p[', x, ',', y, ']');
    write (vt125file, 't''', message, '''')
  end;  (*  vt125text  *)


procedure vt125circle (atpoint : point;
                       radius   : real);

  var
    x, y,
    devrad : integer;

  begin
    x := (round (get_x_coord (atpoint) * vt125max));
    y := (round (get_y_coord (atpoint) * vt125max));
    devrad := round (radius * vt125max);
    write (vt125file, 'p[', x, ',', y, ']');
    write (vt125file, 'c[+', devrad, ']');
  end;  (*  vt125circle  *)


procedure gigiline (p1, p2 : point);

  var
    x, y : integer;

  begin
    (*  ... find the coordinates of the starting position of the
        line and convert them to device coordinates  *)
    x := round (get_x_coord (p1) * gigimax);
    y := round (get_y_coord (p1) * gigimax);
    (*  ... set the current position, for the device, to the start
        of the line  *)
    write (gigifile, 'p[', x, ',', y, ']');
    (*  ... similarly, find the coordinates of the end point of the
        line, and convert them to device coordinates  *)
    x := round (get_x_coord (p2) * gigimax);
    y := round (get_y_coord (p2) * gigimax);
    (*  ... draw a vector on the device, from the current position
        to the end point of the line  *)
    writeln (gigifile, 'v[', x, ',', y, ']')
  end;  (*  gigiline  *)


procedure gigimark (markpoint : point);

  var
    x, y : integer;

  begin
    x := (round (get_x_coord (markpoint) * gigimax)) - 4;
    y := (round (get_y_coord (markpoint) * gigimax)) + 12;
    write (gigifile, 'p[', x, ',', y, ']');
    writeln (gigifile, gigimarker, marksymbol)
  end;  (*  gigimark  *)


procedure gigitext (atpoint  : point;
                    message  : buffer);

  var
    x, y : integer;

  begin
    x := (round (get_x_coord (atpoint) * gigimax));
    y := (round (get_y_coord (atpoint) * gigimax));
    write (gigifile, 'p[', x, ',', y, ']');
    write (gigifile, 't''', message, '''')
  end;  (*  gigitext  *)


procedure gigicircle (atpoint : point;
                      radius  : real);

  var
    x, y,
    devrad : integer;

  begin
    x := (round (get_x_coord (atpoint) * gigimax));
    y := (round (get_y_coord (atpoint) * gigimax));
    devrad := round (radius * gigimax);
    write (gigifile, 'p[', x, ',', y, ']');
    write (gigifile, 'c[+', devrad, ']');
  end;  (*  gigicircle  *)
```

```
procedure vstecline (p1, p2 : point);

  (*  This routine applies to the Cyber version of
      SSOCS
   *)

  var
    x, y : real;

  begin
    (*   ... find the coordinates of the starting position
      of the line, and convert them to device coordinates.  *)
    x := get_x_coord (p1) * vstecmax;
    y := get_y_coord (p1) * vstecmax;
    (*  ... move the versatec pen to the starting position
        of the line

      ... the PLOT routine is not available for the PIXEL version

    plot (x, y, 3);

      ... similarly, find the coordinates of the ending position
      of the line, and convert them to device coordinates.
    *)
    x := get_x_coord (p2) * vstecmax;
    y := get_y_coord (p2) * vstecmax;
    (* ... draw a line from the current position to the
       end point.

    plot (x, y, 2)
     *)
  end;  (*  vstecline  *)


procedure vstectext (atpoint : point;
                     message : buffer);

  var
    x, y : real;

  begin
    x := get_x_coord (atpoint) * vstecmax;
    y := get_y_coord (atpoint) * vstecmax;
    (*  letter (maxchars, vstecletter, Ø.Ø, x, y, message)  *)
  end;  (*  vstectext  *)
```

```
procedure vstecmark (markpoint : point);

  var
    x, y : real;

  begin
    x := (get_x_coord (markpoint) * vstecmax) - vstecmarkoffset;
    y :=  get_y_coord (markpoint) * vstecmax;
    (*  letter (1, vstecletter, 0.0, x, y, '.')  *)
  end;  (*  vstecmark  *)


procedure vsteccircle (centre    : point;
                       ndcradius : real);

  var
    x, y,
    devradius : real;

  begin
    x := get_x_coord (centre) * vstecmax;
    y := get_y_coord (centre) * vstecmax;
    devradius := ndcradius * vstecmax;
    (*  arc (x + devradius, y, x+devradius, y, x, y, 0.001)  *)
  end;  (*  vsteccircle  *)
```

```
(*  DISPLAY PIPELINE PACKAGE  *)


  procedure send_to_display (instance : primitive);

    var
      ok_to_display : boolean;


    procedure map_to_window (var instance : primitive);

    (*  This procedure maps the current output primitive instance to the
        current window;  that is, it performs any rotation of points that
        is required, and clips lines that cross the window boundary.  By
        performing these actions, it determines what information is actually
        visible in the current window.
     *)


      var
        start,
        finish   : point;
        modified : boolean;
        onscreen : switch;


      procedure clip (var startpoint,
                          endpoint    : point;
                      var place       : switch);

        (*  The clipping algorithm used in SSOCS is from Cohen and
            Sutherland, described in Newman and Sproull's "Principles
            of Interactive Computer Graphics", 1979
         *)

        type
          edges = (left, right, bottom, top);
          outcodes = set of edges;

        var
          codeone,                        (*  code values for first point   *)
          codetwo,                        (*  code values for second point  *)
          pcode : outcodes;
          leftbound, rightbound,          (*  left and right window bounds  *)
          upperbound, lowerbound,         (*  upper and lower window bounds *)
          x, y,
          x1, x2,
    y1, y2 : real;


  procedure findcodes (    x, y    : real;
                       var regions : outcodes);

  (*  This procedure determines the code values for the
      point at the specified coordinates
   *)

    begin  (*  findcodes  *)
      regions := [];
      if x < leftbound then
        regions := [left]
      else if x > rightbound then
        regions := [right];
      if y < lowerbound then
        regions := regions + [bottom]
      else if y > upperbound then
        regions := regions + [top]
    end;  (*  findcodes  *)


  begin  (*  clip  *)
    (*
        ... determine the boundaries of the window
     *)
    what_is_window (leftbound, rightbound,
                    lowerbound, upperbound, the_world);
    (*
        ... find the coordinates of the two end points
     *)
    x1 := get_x_coord (startpoint);
    y1 := get_y_coord (startpoint);
    x2 := get_x_coord (endpoint);
    y2 := get_y_coord (endpoint);
    (*
        ... get the code values for the end points
     *)
    findcodes (x1, y1, codeone);
    findcodes (x2, y2, codetwo);
    (*
        ... clip the points to the window, if necessary
     *)
    while ((codeone <> []) or (codetwo <> [])) and
          (codeone * codetwo = []) do
      begin
        pcode := codeone;
```

```
          if pcode = [] then
            pcode := codetwo;
          if left in pcode then  (*  line crosses left edge  *)
            begin
              y := y1 + (y2 - y1) * (leftbound - x1) / (x2 - x1);
              x := leftbound
            end
          else if right in pcode then  (*  line crosses right edge  *)
            begin
              y := y1 + (y2 - y1) * (rightbound - x1) / (x2 - x1);
              x := rightbound
            end
          else if bottom in pcode then  (*  line crosses bottom edge  *)
            begin
              x := x1 + (x2 - x1) * (lowerbound - y1) / (y2 - y1);
              y := lowerbound
            end
          else if top in pcode then  (*  line crosses top edge  *)
            begin
              x := x1 + (x2 - x1) * (upperbound - y1) / (y2 - y1);
              y := upperbound
            end;
          if pcode = codeone then
            begin
              x1 := x;
              y1 := y;
              findcodes (x, y, codeone)
            end
          else
            begin
              x2 := x;
              y2 := y;
              findcodes (x, y, codetwo)
            end
        end;
      if codeone * codetwo <> [] then
        place := off
      else
        begin
          place := on;
          set_coordinates (x1, y1, startpoint);
          set_coordinates (x2, y2, endpoint)
        end
    end;  (*  clip  *)


  function whereispoint (thepoint : point) : switch;
```

```
        (*  this function is used to determine if a point lies
            inside the window
         *)

        var
          wxl, wxr,
          wyb, wyt : real;

        begin  (*  whereispoint  *)
          what_is_window (wxl, wxr, wyb, wyt, the_world);
          if (get_x_coord (thepoint) >= wxl) and
             (get_x_coord (thepoint) <= wxr) and
             (get_y_coord (thepoint) >= wyb) and
             (get_y_coord (thepoint) <= wyt) then
            whereispoint := on
          else
            whereispoint := off
        end;  (*  whereispoint  *)


      begin  (*  map_to_window  *)
        case get_the_action (instance) of
          line :
            begin
              (*
                 ... the current instance is a line.  The end points of the
                 line are retrieved, and rotation or clipping is performed,
                 if necessary
               *)
              get_end_points (start, finish, instance);
              modified := false;
              the_line_is (on, instance);
              if get_angle (the_world) <> 0.0 then
                begin
                  rotate_point (start, get_angle (the_world));
                  rotate_point (finish, get_angle (the_world));
                  modified := true
                end;
              if clip_switch (the_world) = on then
                begin
                  clip (start, finish, onscreen);
                  modified := true;
                  the_line_is (onscreen, instance)
                end;
              (*
                 ... if either rotation or clipping has been performed it is
                 assumed that the end points of the line have been changed,
                 so their values in "instance" are updated
             *)
            if line_is_on (instance) and
               modified then
              set_end_points (start, finish, instance)
          end;
        marker :
          begin
            what_is_action_point (start, instance);
            modified := false;
            the_point_is (on, instance);
            if get_angle (the_world) <> 0.0 then
              begin
                rotate_point (start, get_angle (the_world));
                modified := true
              end;
            if clip_switch (the_world) = on then
              the_point_is (whereispoint (start), instance);
            if point_is_on (instance) and
               modified then
              go_to_action_point (start, instance)
          end
      end
    end;  (*  map_to_window  *)


  procedure map_to_viewport (var instance : primitive);

  (*  This procedure transforms the coordinates, specified in the
      instance of the current primitive, from the world coordinate
      system to the corresponding ndc space visible in the current
      viewport.
   *)

    var
      start,
      finish : point;

    begin
      case get_the_action (instance) of
        line :
          begin
            (*
                ... the current output primitive is a line.  The end
                points of the line are retrieved, transformed to
                the visible ndc space, and replaced in the variable
                describing the current instance
             *)
            get_end_points (start, finish, instance);
            trans_world_to_ndc (start);    (*  for starting point of line  *)
            trans_world_to_ndc (finish);   (*  for end point of line  *)
            set_end_points (start, finish, instance)
          end;
        marker :
          begin
            (*
                ... the current output primitive is a marker or text.
                The position of the primitive is retrieved, transformed
                to the visible ndc space, and replaced in the variable
                describing the current instance
             *)
            what_is_action_point (finish, instance);
            trans_world_to_ndc (finish);
            go_to_action_point (finish, instance)
          end
      end
    end;  (*  map_to_viewport  *)


  procedure dispatch (instance : primitive);


    var
      start,
      finish      : point;
      display     : devices;
      charstring  : buffer;

    begin  (*  dispatch  *)
      (*
         ... determine the kind of output primitive
       *)
      case get_the_action (instance) of
        line :
          begin
            (*
                ... the current output primitive is a line.  Find the
                end points of this line, and for each device in the
                set of selected_surfaces, execute the appropriate
                line-drawing routine
             *)
            get_end_points (start, finish, instance);
            for display := vt125 to prism do
              if display in selected_surfaces then
                case display of
                  vt125 :
                    vt125line (start, finish);
                prism :
                  (*  prismline (start, finish)  *);
                gigi :
                  gigiline (start, finish);
                versatec :
                  (*  THIS ROUTINE IS NOT TO BE CALLED FOR THIS VERSION
                  vstecline (start, finish)  *);
              end
        end;  (*  of line section  *)
      marker :
        begin
          what_is_action_point (start, instance);
          for display := vt125 to prism do
            if display in selected_surfaces then
              case display of
                vt125 :
                  vt125mark (start);
                prism :
                  (*  prismmark (start)  *);
                gigi :
                  gigimark (start);
                versatec :
                  (*  THIS ROUTINE IS NOT TO BE CALLED FOR THIS VERSION
                  vstecmark (start)  *);
              end
        end;  (*  of marker section  *)
      escapade :
        case get_function (instance) of
          prompt :
            begin
              what_is_action_point (start, instance);
              get_message (charstring, instance);
              case get_surface (instance) of
                vt125 :
                  vt125text (start, charstring);
                prism :
                  (*  prismtext (start, charstring);  *);
                gigi :
                  gigitext (start, charstring);
                versatec :
                  (*  THIS ROUTINE IS NOT TO BE CALLED
                  vstectext (start, charstring)  *);
              end
            end;  (*  of prompt section  *)
          circle :
            begin
              what_is_action_point (start, instance);
              case get_surface (instance) of
```

```
                  vt125 :
                    vt125circle (start, get_radius (instance));
                  prism :
                    (*  prismcircle (start, get_radius (instance))  *);
                  gigi :
                    gigicircle (start, get_radius (instance));
                  versatec :
                    (*  THIS ROUTINE IS NOT TO BE CALLED ...
                    vsteccircle (start, get_radius (instance))  *);
                end
              end  (*  of circle section  *)
          end  (*  of escapade section  *)
      end
    end;  (*  dispatcher  *)


  begin  (*  send_to_display  *)
    if get_the_action (instance) = escapade then
      dispatch (instance)
    else
      begin
        map_to_window (instance);
        case get_the_action (instance) of
          line :
            ok_to_display := line_is_on (instance);
          marker :
            ok_to_display := point_is_on (instance)
        end;
        if ok_to_display then
          begin
            map_to_viewport (instance);
            dispatch (instance)
          end
      end
  end;  (*  send_to_display  *)
```

```
(*  OUTPUT PRIMITIVE FUNCTIONS  *)


(*  moves *)


  procedure move_abs (x, y : real);

  (*  The cp is set to the position (x, y), where x and y are coordinates
      in world space.  Invocation of this procedure merely sets the cp;
      no drawing commands are output.
  *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else
        set_coordinates (x, y, cp)
    end;  (*  move_abs  *)


  procedure move_rel (dx, dy : real);

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else
        set_coordinates (dx + get_x_coord (cp), dy + get_y_coord (cp), cp)
    end;  (*  move_rel  *)


  procedure query_cp (var x, y : real);

  (*  The current drawing position is copied into the parameters
      x and y.
  *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else
        begin
          x := get_x_coord (cp);
          y := get_y_coord (cp)
      end
    end;  (*  query_cp  *)


(*  lines  *)


  procedure line_abs (x, y : real);

  (*  This procedure is used to describe a line of an object in world
      coordinates.  The line runs from the current position (cp) to the
      position specified by (x, y).

      If the position specified is coincident with the cp, the appearance
      of the line is device-dependent (in this implementation, no further
      action is taken, i.e. the line_abs procedure exits immediately).

      The cp is updated to (x, y).
   *)

    var
      instance : primitive;
      endpoint : point;

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if not segment_is_open (controls) then
        process_error (201, 6)
      else
        begin
          (*
              ... the endpoint of the line is defined by the input
              coordinates, and it begins at the cp.  If both of these
              points are equal then nothing else is done and the procedure
              will exit
           *)
          set_coordinates (x, y, endpoint);
          if not points_equal (cp, endpoint) then
            begin
              (*
                  ... since this is a valid line (i.e. it is accepted
                  for display), the "instance" of this current primitive
                  (the line) is described, and the cp is updated to the
                  end point.
               *)
            set_action_to (line, instance);
            set_end_points (cp, endpoint, instance);
            send_to_display (instance);
            cp := endpoint
          end
      end
  end;  (*  line_abs  *)


procedure line_rel (dx, dy : real);

  begin
    line_abs (dx + get_x_coord (cp), dy + get_y_coord (cp));
  end;  (*  line_rel  *)



(*  polylines  *)


procedure poly_abs (x_array, y_array : coordseq;
                    n                : coordrange);

  var
    i : integer;

  begin
    for i := 1 to n do
      line_abs (x_array [i], y_array [i])
  end;  (*  poly_abs  *)


procedure poly_rel (dx_array, dy_array : coordseq;
                    n                  : coordrange);

  var
    i : integer;

  begin
    for i:= 1 to n do
      line_abs (dx_array [i] + get_x_coord (cp),
                dy_array [i] + get_y_coord (cp))
  end;  (*  poly_rel  *)



(*  markers  *)
```

```
procedure marker_abs (x, y : real);

(*  The cp is updated to (x, y), and then the marker symbol is created.
    The marker is centred at the transformed position of the cp.
*)

  var
    instance : primitive;

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else if not segment_is_open (controls) then
      process_error (201, 6)
    else
      begin
        (*
            ... the cp is updated to the point specified by the parameters
            x and y.  The "instance" of the marker is then prepared, and
            sent for display.
         *)
        set_coordinates (x, y, cp);
        set_action_to (marker, instance);
        go_to_action_point (cp, instance);
        send_to_display (instance)
      end
  end;  (*  marker_abs  *)


procedure marker_rel (dx, dy : real);

  begin
    marker_abs (dx + get_x_coord (cp), dy + get_y_coord (cp));
  end;  (*  marker_rel  *)


(*  escape functions  *)


  procedure escape (name      : validescapes;
                    parmno    : integer;
                    parmlist  : escapestuff);

    var
      instance : primitive;
```

```
function validparmlist (thelist : escapestuff) : boolean;

  var
    okay      : boolean;
    width,
    height    : real;
    p         : point;

  begin
    okay := true;
    if not (sel_surface (thelist) in init_surfaces) then
      okay := false
    else
      begin
        what_is_ndc_space (width, height, the_world);
        if sel_function (thelist) in esccodes then
          case sel_function (thelist) of
            prompt :
              begin
                sel_prompt_start (p, thelist);
                okay := (get_x_coord (p) >= Ø.Ø) and
                        (get_x_coord (p) <= width) and
                        (get_y_coord (p) >= Ø.Ø) and
                        (get_y_coord (p) <= height)
              end;
            circle :
              begin
                sel_centre (p, thelist);
                okay := (get_x_coord (p) >= Ø.Ø) and
                        (get_x_coord (p) <= width) and
                        (get_y_coord (p) >= Ø.Ø) and
                        (get_y_coord (p) <= height)
              end
          end
        else
          process_error (1ØØØ, 6);
        validparmlist := okay
      end
  end;  (*  invalidparmlist  *)


begin
  check_call_is_valid (controls);
  if core_is_down (controls) then
    process_error (717, 6)
  else if not validparmlist (parmlist) then
```

```
          process_error (8Ø3, 5)
        else
          begin
            set_action_to (escapade, instance);
            set_parameters (parmlist, instance);
            send_to_display (instance)
          end
      end;  (*  escape  *)
```

```
(*  SEGMENT OPERATION PACKAGE  *)

(*

 Temporary segments are provided for use by application programs or
 portions of application programs that do not require the picture
 modification capabilities of retained segments (Retained segments are
 not yet implemented).  Although all output primitives must be included
 within a segment, SSOCS does not retain the image if the output prim-
 itives are part of a temporary segment.

 Temporary segments conceptually represent a very simple graphical data
 structure.  Only two operations can be performed on the picture:

       1.  temporary segments can be added, and

       2.  all temporary segments can be deleted.

 Temporary segments have no names and no attributes.  It is impossible to
 explicitly remove the images of individual temporary segments from a view
 surface.

 *)


 procedure create_temporary_segment ;

 (*  This procedure creates a new, empty, temporary segment.  The temporary
     segment becomes the open segment.  Subsequent output primitive function
     invocations will result in new information appearing on the currently
     selected view surface(s).
  *)

 begin
   check_call_is_valid (controls);
   if core_is_down (controls) then
     process_error (717, 6)
   else if selected_surfaces = [] then
     process_error (4, 6)
   else if segment_is_open (controls) then
     process_error (301, 6)
   else
     begin
       a_segment_is (ouvert, controls);
       check_ndc_space_state
     end
```

```
end;  (*  create_temporary_segment  *)


procedure close_temporary_segment ;

(*  The currently open temporary segment becomes closed; output
    primitives can no longer be sent to the selected view surface(s).
 *)

begin
  check_call_is_valid (controls);
  if core_is_down (controls) then
    process_error (717, 6)
  else if segment_is_open (controls) then
    a_segment_is (ferme, controls)
  else
    process_error (3Ø7, 6)
end;  (*  close_temporary_segment  *)


procedure query_temporary_segment (var open : condition);

(*  The temporary segment status of SSOCS is copied into 'open'.
 *)

begin
  check_call_is_valid (controls);
  if core_is_down (controls) then
    process_error (717, 6)
  else if segment_is_open (controls) then
    open := ouvert
  else
    open := ferme
end;  (*  query_temporary_segment  *)
```

```
(*   VIEW SURFACE PACKAGE

  Two functions are provided to add view surfaces to, and remove
  view surfaces from, the set of selected view surfaces.  When a
  segment is created, it is associated with the surfaces that are
  currently selected, so that, afterwards, until the segment is
  deleted, the image of the segment appears only on those view
  surfaces.  The set of selected view surfaces cannot be changed
  while there is an open segment.

*)


  procedure select_view_surface (surfacename : devices);

  (*
     This procedure adds the view surface 'surfacename' to the set
     of selected view surfaces.  The set of selected view surfaces is
     used for two purposes:

           1.  When a segment is created, the image of the segment appears
               only on those surfaces in the set of selected view surfaces.
           2.  When new_frame is called, a new-frame action occurs only
               on view surfaces in the set of selected view surfaces.

     The set of selected view surfaces is initially empty.
   *)

  begin
    check_call_is_valid (controls);
    if core_is_down (controls) then
      process_error (717, 6)
    else if segment_is_open (controls) then
      process_error (6, 6)
    else if not (surfacename in init_surfaces) then
      process_error (708, 6)
    else if surfacename in selected_surfaces then
      process_error (709, 6)
    else
      selected_surfaces := selected_surfaces + [surfacename]
  end;  (*  select_view_surface  *)


  procedure deselect_view_surface (surfacename : devices);

   (*
    This procedure removes the view surface 'surfacename' from the
    set of selected view surfaces.  Subsequent segment creations and new
    frame invocations will not affect this view surface until it is
    reselected with a select_view_surface invocation.
   *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if segment_is_open (controls) then
        process_error (6, 6)
      else if not (surfacename in selected_surfaces) then
        process_error (711, 6)
      else
        selected_surfaces := selected_surfaces - [surfacename]
    end;  (*  deselect_view_surface  *)


  (*

  A view surface must be initialized before it can be used.
  The procedures select_view_surface and deselect_view_surface add view
  surfaces to, and remove view surfaces from, the set of selected
  view surfaces.

  The set of selected view surfaces cannot be changed while there is an open
  segment.  View surface selection only affects the create_temporary_segment
  and new_frame procedures.

  *)


  procedure init_view_surface (surfacename : devices);

   (*
    This procedure adds the specified view surface surfacename to
    the set of initialized view surfaces, and initializes that surface.
    The view surface must be initialized with the init_view_surface
    procedure before it can be selected for graphic output with the
    select_view_surface procedure.  The init_view_surface procedure does
    not perform an implicit select_view_surface for the specified surface.
   *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if surfacename in init_surfaces then
```

```
        process_error (705, 6)
      else
        begin
          (*
              ... the specified surface is added to the set of initialized
              view surfaces
           *)
          init_surfaces := init_surfaces + [surfacename];
          (*
              ... the specified device is "connected" to the application
              program, and prepared for graphics output
           *)
          make_device (access, surfacename);
          make_device (startup, surfacename);
        end
    end;  (*  init_view_surface  *)


  procedure term_view_surface (surfacename : devices);

  (*
    This procedure terminates access to the view surface surfacename.
    Segments whose images appear on only this surface are deleted.

    In this implementation, the view surface is cleared.
   *)

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if not (surfacename in init_surfaces) then
        process_error (708, 6)
      else
        begin
          init_surfaces := init_surfaces - [surfacename];
          make_device (clear, surfacename);
          make_device (finishoff, surfacename)
        end
    end;  (*  term_view_surface  *)


  procedure query_selected_surfaces (     arraysize        : integer;
                                      var viewsurfacenames : surarray;
                                      var numberofsurfaces : integer);

    var
      display : devices;
    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else  if arraysize <= Ø then
        process_error (3, 5)
      else
        begin
          numberofsurfaces := Ø;
          for display := vt125 to prism do
            if display in selected_surfaces then
              begin
                if arraysize > Ø then
                  begin
                    viewsurfacenames [arraysize] := display;
                    arraysize := arraysize - 1
                  end;
                numberofsurfaces := numberofsurfaces + 1
              end
        end
    end;  (*  query_selected_surfaces  *)


  procedure new_frame ;

  (*
    ...this procedure causes a new-frame action for each of the
    currently selected view surfaces.
   *)

    var
      display : devices;

    begin
      check_call_is_valid (controls);
      if core_is_down (controls) then
        process_error (717, 6)
      else if selected_surfaces = [] then
        process_error (4, 6)
      else
        for display := vt125 to prism do
          if display in selected_surfaces then
            make_device (clear, display)
    end;  (*  new_frame  *)
```

```
(*  INITIALIZATION MODULE

  Since variables cannot be initialized at compile time in Pascal, a
  special routine MUST be called before any other SSOCS routine, in
  order to guarantee the initial state of the library

 *)


  procedure reset_values ;

    begin
      core_is (down, controls);                       (*  system is down        *)
      a_segment_is (ferme, controls);                 (*  no segment open       *)
      set_error_processing (off, controls);           (*  no errors             *)
      set_report_flushed (true, controls);            (*  no error reports      *)
      the_window_is (down, controls);                 (*  window is down        *)
      the_viewport_is (down, controls);               (*  viewport is down      *)
      ndc_is (down, controls);                        (*  ndc space is down     *)
      set_ndc_default_state (down, controls);         (*  ndc default no set    *)
      selected_surfaces := [];                        (*  no surfaces going     *)
      init_surfaces := [];                            (*  no surfaces set up    *)
      rewrite (stderr, erroractual);                  (*  set up error log      *)
      primcodes := [line, marker, escapade];          (*  possible actions      *)
      esccodes  := [prompt, circle];                  (*  possible escapes      *)
    end;  (*  reset_values  *)
```

```
(*  INITIALIZATION AND TERMINATION OF THE SSOCS LIBRARY
 *)


  procedure initialize_core ;

  (*
    initialize_core must be the first SSOCS procedure invoked (after a
    call to reset_values).  It guarantees that the library is in a
    predefined state, with the default settings of all the SSOCS
    parameters established.
   *)

    begin
      if core_is_down (controls) then
        begin
          reset_values ;
          core_is (up, controls);
          set_window (0.0, 1.0, 0.0, 1.0);
          set_clipping (on);
          set_view_up (0.0, 1.0);
          move_abs (0.0, 0.0);
        end
      else
        process_error (701, 6)
    end;  (*  initialize_core  *)


  procedure terminate_core ;

  (*
    This procedure closes any open segment, terminates all initial-
    ized view surfaces, and releases all other resources being used by
    the SSOCS system.  This procedure should be used to terminate the
    use of the library, and may be invoked at any time after the SSOCS
    system is initialized.  After the library is terminated, it may
    be reinitialized with the initialize_core procedure.
   *)

    var
      open_surface : devices;

    begin
      if core_is_down (controls) then
        process_error (717, 6)
      else
        if segment_is_open (controls) then
```

```
          a_segment_is (ferme, controls);
        if init_surfaces <> [] then
          for open_surface := vt125 to prism do
            if open_surface in init_surfaces then
              begin
                make_device (clear, open_surface);
                make_device (finishoff, open_surface);
                init_surfaces := init_surfaces - [open_surface]
              end;
        reset_values
    end;  (*  terminate_core  *)
```

```
(*  SAMPLE APPLICATION PROGRAM  *)


  procedure box (xl, yl, width, height : real);

    begin
      move_abs (xl, yl);
      line_rel (width, Ø.Ø);
      line_rel (Ø.Ø, height);
      line_rel (-width, Ø.Ø);
      line_abs (xl, yl)
    end;  (*  box  *)


  procedure arrow (xl, yl, length : real);

    (*  draws a right arrow  *)

    begin
      move_abs (xl, yl);
      line_abs (xl+length-Ø.2, yl);
      move_abs (xl+length, yl);
      line_rel (-Ø.2, Ø.1);
      line_rel (Ø.Ø, -Ø.2);
      line_abs (xl+length, yl)
    end;  (*  arrow  *)


  procedure chair (x, y : real);

    begin
      move_abs (x, y);
      line_rel (2.5, Ø.Ø);
      line_rel (Ø.Ø, 1.Ø);
      line_rel (-Ø.75, 1.Ø);
      line_rel (-1.Ø, Ø.Ø);
      line_rel (-Ø.75, -1.Ø);
      line_abs (x, y);
    end;  (*  chair  *)


  procedure desktop;

    begin
      move_abs (1Ø.Ø, 1Ø.Ø);
      move_rel (2.5, Ø.2);
      line_rel (2.5, Ø.Ø);
      line_rel (Ø.Ø, 2.Ø);
      line_rel (-2.5, Ø.Ø);
      line_rel (Ø.Ø, -2.Ø)
    end;


  procedure diary;

    begin
      move_abs (12.5, 12.5);
      line_rel (1.Ø, Ø.5);
      line_rel (-Ø.25, Ø.5);
      line_rel (-1.Ø, -Ø.5);
      line_rel (Ø.25, -Ø.5)
    end;  (*  diary  *)


  procedure plan;

  (*  draw floor plan  *)

    begin
      move_abs (5.Ø, 5.Ø);
      line_abs (45.Ø, 5.Ø);
      line_abs (45.Ø, 45.Ø);
      line_rel (-2Ø.Ø, Ø.Ø);
      line_rel (Ø.Ø, -1Ø.Ø);
      line_rel (-2Ø.Ø, Ø.Ø);
      line_abs (5.Ø, 5.Ø);

      move_rel (15.Ø, Ø.Ø);
      line_rel (Ø.Ø, 2Ø.Ø);
      line_rel (-15.Ø, Ø.Ø);

      move_rel (15.Ø, Ø.Ø);
      line_rel (5.Ø, Ø.Ø);
      line_rel (Ø.Ø, 1Ø.Ø);
      move_rel (Ø.Ø, -1Ø.Ø);
      line_rel (Ø.Ø, -5.Ø);
      line_rel (2Ø.Ø, Ø.Ø);
      move_rel (-15.Ø, Ø.Ø);
      line_rel (Ø.Ø, -15.Ø);
      move_abs (25.Ø, 3Ø.Ø);
      line_rel (2Ø.Ø, Ø.Ø);

      move_abs (5.Ø, 7.5);
      line_rel (2.5, Ø.Ø);
      line_rel (Ø.Ø, 1Ø.Ø);
      line_rel (-2.5, Ø.Ø);
      box (1Ø.Ø, 1Ø.Ø, 7.5, 5.Ø);
      chair (12.5, 7.5);
      move_abs (1Ø.Ø, 25.Ø);
      line_rel (Ø.Ø, -2.5);
      line_rel (7.5, Ø.Ø);
      line_rel (2.5, Ø.Ø);

      desktop;
      diary;
    end;  (*  plan  *)


begin

  (*  this routine must always be called before anything
      else is done!!!!  *)
  reset_values;

  initialize_core ;
  init_view_surface (gigi);
  select_view_surface (gigi);
  set_window (Ø.Ø, 8.Ø, Ø.Ø, 8.Ø);
  create_temporary_segment;
  box (Ø.Ø, 2.25, 1.25, 1.5);
  box (2.Ø, 1.Ø, 5.Ø, 4.Ø);
  box (2.5, 2.25, 1.5, 1.5);
  box (4.75, 2.Ø, Ø.5, 2.25);
  box (5.75, 2.25, Ø.75, Ø.75);
  box (5.75, 3.25, Ø.75, Ø.75);
  arrow (1.25, 3.3, 1.25);
  arrow (1.25, 2.6, 1.25);
  arrow (4.Ø, 3.Ø, Ø.75);
  arrow (5.25, 2.6, Ø.5);
  arrow (5.25, 3.6, Ø.5);
  arrow (6.5, 2.6, 1.Ø);
  arrow (6.5, 3.6, 1.Ø);
  close_temporary_segment;

  writeln (' ready for # 2?');
  readln (answer);

  term_view_surface (gigi);
  deselect_view_surface (gigi);
  terminate_core;
```

```
(* number 2 diagram *)

  initialize_core;
  init_view_surface (gigi);
  select_view_surface (gigi);
  set_window (Ø.Ø, 5Ø.Ø, Ø.Ø, 5Ø.Ø);
  create_temporary_segment;

  plan;

  close_temporary_segment;

  writeln (' ready to proceed? ');
  readln (answer);
  new_frame;

  (* part of plan *)

  set_window (Ø.Ø, 25.Ø, Ø.Ø, 27.Ø);
  create_temporary_segment;

  plan;

  close_temporary_segment;
  writeln (' ok? ');
  readln (answer);
  new_frame;

  (* one room *)

  set_window (5.Ø, 2Ø.Ø, 5.Ø, 25.Ø);
  set_viewport (Ø.Ø, Ø.75, Ø.Ø, 1.Ø);
  create_temporary_segment;

  plan;

  close_temporary_segment;
  writeln (' ok? ');
  readln (answer);
  new_frame;

  (* part of the desk *)

  set_window (11.Ø, 18.Ø, 1Ø.Ø, 14.Ø);
  set_viewport (Ø.Ø, 1.Ø, Ø.Ø, Ø.571);
  create_temporary_segment;

  plan;
```

```
close_temporary_segment;

(*  now different viewports  *)

writeln (' ready to go on?');
readln (answer);
new_frame;
set_window (Ø.Ø, 5Ø.Ø, Ø.Ø, 5Ø.Ø);
set_viewport (Ø.Ø, Ø.5, Ø.5, 1.Ø);
create_temporary_segment;

plan;  (*  complete floor  *)

close_temporary_segment;

writeln (' ready to proceed? ');
readln (answer);

(*  part of plan  *)

set_window (Ø.Ø, 25.Ø, Ø.Ø, 27.Ø);
set_viewport (Ø.5, 1.Ø, Ø.5, Ø.9);
create_temporary_segment;

plan;

close_temporary_segment;
writeln (' ok? ');
readln (answer);

(*  one room  *)

set_window (5.Ø, 2Ø.Ø, 5.Ø, 25.Ø);
set_viewport (Ø.Ø62, Ø.437, Ø.Ø, Ø.5);
create_temporary_segment;

plan;

close_temporary_segment;
writeln (' ok? ');
readln (answer);

(*  part of the desk  *)

set_window (11.Ø, 18.Ø, 1Ø.Ø, 14.Ø);
set_viewport (Ø.5, 1.Ø, Ø.1Ø8, Ø.392);
create_temporary_segment;
```

```
  plan;

  move_abs (Ø.Ø, Ø.Ø);
  close_temporary_segment;
  terminate_core
end.  (*  main program  *)
```

REFERENCES

BØ80	Bø, K., "Standardisation of Graphics Software", Computer Graphics: Invited Papers, B. Shackel (ed), Infotech International Limited, 1980. pp. 82 - 93.
DOHE79	Doherty, W.J., "The Commercial Significance of Man/Computer Communication", Man/Computer Communication: Invited Papers, B. Shackel (ed), Infotech International Limited, 1979. pp. 81 - 93.
ENCA76	Encarnacao, J., and G. Nees, "Recommendations on Methodology in Computer Graphics", Methodology in Computer Graphics, Guedj, R.A., and H.A. Tucker (eds), North-Holland, 1976, pp. 9 - 26.
FOLE76	Foley, J.D., "Output Primitives", Methodology in Computer Graphics, Guedj, R.A., and H.A. Tucker (eds), North-Holland, 1976, pp. 119 - 122.
FOLE81	Foley, J.D. and P.A. Wenner, "The George Washington University Core System Implementation", Computer Graphics, Vol. 15, No. 3, August 1981, pp. 123 - 131.
FRIE79	Frieden, A, "A CORE Viewing System for APL", Computer Graphics, Vol. 13, No. 1, 1979, pp. 55 - 77.
GSPC77	"Status Report of the Graphics Standards Planning Committee", published as Computer Graphics, Vol. 11, No. 3, Fall 1977.
GSPC79	"Status Report of the Graphics Standards Planning Committee", published as Computer Graphics, Vol. 13, No. 3, August 1979.
GUED76	Guedj, R.A., "Some Methodological Remarks for the Workshop", Methodology in Computer Graphics, Guedj, R.A., and H.A. Tucker (eds), North-Holland, 1976, pp. 3 - 8.

HOPG76 Hopgood, F.R.A., "Is a Graphics Standard Possible?", Methodology in Computer Graphics, Guedj, R.A., and H.A. Tucker (eds), North-Holland, 1976, pp. 9 - 26.

LISK77 Liskov, B. *et al*, "Abstraction Mechanisms in CLU", Communications of the ACM, Vol. 20, No. 8, August 1977, pp. 546 - 576.

MICH78 Michener, J.C. and J.D. Foley, "Some Major Issues in the Design of the Core System", Computing Surveys, Vol. 10, No. 4, December 1978, pp. 445 - 463.

NEWM74 Newman, W.M. and R.F. Sproull, "An Approach to Graphics System Design", Proceedings of the I.E.E.E., Vol. 62, No. 4, April 1974, pp. 471-483.

NEWM78 Newman, W.M. and A. van Dam, "Recent Efforts Toward Graphics Standardization", Computing Surveys, Vol. 10, No. 4, December 1978, pp. 365-380.

NEWM79 Newman, W.M. and R.F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, 1979, pp.63 - 76.

NICO81 Nicol, C.J. and A.C. Kilgour, "A Pascal Implementation of the GSPC Core Graphics Package", Computer Graphics, Vol. 15, No. 4, December 1981, pp.327 - 335.

SANC76 Sancha, T.L., "Guidelines for the IFIP Workshop on Graphics Methodology", Methodology in Computer Graphics, Guedj, R.A., and H.A. Tucker (eds), North-Holland, 1976, pp. 123 - 126.

STEW79 Stewart, T.F.M., "Visual Communication", Man/Computer Communication: Invited Papers, B. Shackel (ed), Infotech International Limited, 1979. pp. 82 - 93.

STLU82 Stluka, F.P. *et al*, "Overview of the University of Pennsylvania CORE System", Computer Graphics, Vol. 16, No. 2, June 1982, pp. 177 - 186.

SUTH70 Sutherland, I.E., "Computer Displays", Scientific American, Vol. 226, No. 6, June 1970, pp. 56 - 80.

WARN78 Warner, J.R., Polisher, M.A. and Kopolow, R.N., "DIGRAF - A FORTRAN Implementation of the Proposed GSPC Standard", Computer Graphics, Vol. 12, No. 3, August 1978, pp. 301 - 307.