DESIGN AND IMPLEMENTATION OF A 2D GRAPHICS PACKAGE

IN ADA 95

By

LAN LI

Bachelor of Science

ZheJiang University

HangZhou, China

1987

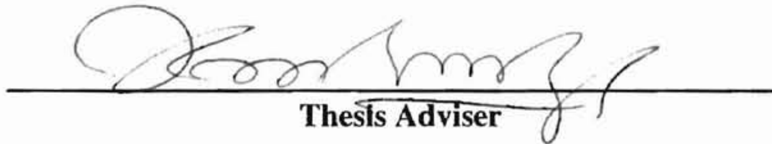
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1996

OKLAHOMA STATE UNIVERSITY

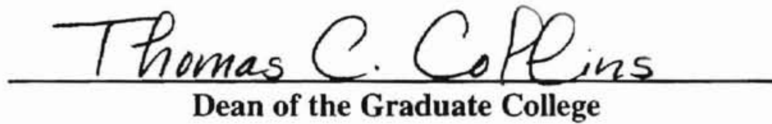
DESIGN AND IMPLEMENTATION OF A 2D GRAPHICS PACKAGE

IN ADA 95

Thesis approved:



Thesis Adviser



Dean of the Graduate College

OKLAHOMA STATE UNIVERSITY

ACKNOWLEDGMENT

I express my sincere gratitude to my advisor Dr. George for his constructive guidance, supervision, inspiration and financial support. Without his understanding and support, I could not have accomplished this project. Appreciation also extends to my committee members Dr. Chandler and Dr. Lu, their great help are invaluable when I was in difficult situation.

I would also like to thank my husband who always gives me encouragement and support in the background. Thanks my son Eric, who was just born, for sharing my happiness and pain, tolerating my occasional inattention during this project. My special thanks also go to my parents who help me take care of my baby with their tremendous love that give me much time to finish this work.

This project is supported by DISA Grant DCA100-96-1-0007

OKLAHOMA STATE UNIVERSITY

TABLE OF CONTENTS

Chapter Page

1. Introduction..1

2. Literature Review 3

2.1. A Review of Standard Graphics Packages. 3

2.1.1. GKS (Graphical Kernel System).. 3

a) Logical Workstations 5

b) Graphics Primitives.. 6

c) logical Input Devices... 7

d) Mode of Interaction... 7

e) Segmentation 7

f) Metafile... 8

2.1.2. PHIGS (Programmer's Hierarchical Interactive Graphics System). 8

a) Graphics Output 9

b) Graphics Input.. 10

c) Interaction handling 10

d) Storage.. 10

2.1.3. SRGP (Simple Raster Graphics Package).. 13

a) Drawing Primitives... 13

OKLAHOMA STATE UNIVERSITY

Chapter	Page

b) Drawing Attributes 13

c) Interaction Handling 13

d) Limitations of SRGP 15

2.2. Review of Ada 95. 15

Packages 16

Private types 16

Operator overloading 17

User-defined *exception* 17

Genericity 17

Encapsulation 18

Inheritance 18

Polymorphism 18

3. GNAT 95 Graphics Package Design and Implement Issues 19

3.1. Software Dependency in GNA95GP. 19

3.2. General Approach to GNA95GP Design. 20

3.3. Design and Implementation of Graphics primitives 21

3.3.1. Primitives and Attributes. 21

3.3.2. Implementation of graphics primitives. 21

3.4. Design of interactive handling. 27

3.5. Storage of graphics primitives. 28

3.5.1. Design of storage scheme. 29

OKLAHOMA STATE UNIVERSITY

Chapter | Page

3.5.2. Implementation of storage scheme........ 35

4. Summary........ 38

5. Bibliography........ 39

6. Appendix I........ 42

7. Appendix II........ 47

8. Appendix III........ 73

OKLAHOMA STATE UNIVERSITY

LIST OF TABLES

Table **Page**

1. Output primitives and descriptions.....22

2. Attribute routines and descriptions.....22

3. Package names.....23

OKLAHOMA STATE UNIVERSITY

LIST OF FIGURES

Figure **Page**

1. Layer mode representation in GKS....5

2. Description of a workstation....6

3. Example of a graphics object storing in PHIGS-CSS....12

4. PHIGS organization model....12

5. Sampling verus event-handling using event queue....15

6. Software dependency in GNA95GP....19

7. Components inside GNA95GP....21

8. Design of drawing-packages using Booch diagram....23

9. Sample output....24

10. Communication scheme used between C and Ada programming....26

11. Example-- Implementation of a primitive....26

12. Interaction handling model in GNA95GP....28

13. An example structure....30

14. An example of attribute rule applied to primitives....31

15. Code segment used to edit TREE-STRUCTURE....33

16. Sequence of TREE_STRUCTURE during the editing....34

17. Global linked list for implementation of storage scheme....35

18. Data structures for structure_node and element_node....37

OKLAHOMA STATE UNIVERSITY

1. Introduction

Several graphics libraries and packages are available as public or proprietary software. Some of these are drawing packages targeted for specific applications, others are designed to be used with high-level programming languages. Previous experience suggests that general purpose packages such as X-Windows are difficult for novices to learn. In order to use these libraries effectively one needs to spend much time learning how to do windows programming and to be familiar with many of the interface functions provided in the libraries. Even though there are several graphics packages available such as GKS (Graphics Kernel System) [15,16,17], PHIGS (Programmer's Hierarchical Interactive Graphics System), and SRGP (Simple Raster Graphics Package) [19,20,21], to the best of our knowledge there is no Ada 95 graphics package available in the public domain. Most of the publicly available libraries are designed to be used with C. Ada 95 is an object-oriented programming language; currently, it is the only object-oriented programming language accepted as an ISO standard. As an object-oriented programming language, it supports reusability and extensibility. There are large application areas that use Ada 95 as the primary programming language. Graphics applications can be found in several of these application areas such as simulation and visualization. Therefore, a graphics library package to be used with Ada 95 is useful. The goal of this thesis is to develop an Ada 95 graphics package that is easy to use and to learn. The graphics package is called GNA95GP.

OKLAHOMA STATE UNIVERSITY

GNA95GP focuses on the development of tools used in undergraduate Computer Graphics courses. GNA95GP is a free software which is modeled after SRGP and SPHIGS to provide a high-level programming interface for students to write graphics applications in Ada 95. It is designed to support dynamic and interactive Ada 95 graphics applications which run under Microsoft Windows 3.1 in IBM PCs.

In Chapter 2, we review several graphics packages and the Ada 95 language with respect to its features that support Object-Oriented and reusable software. In Chapter 3, we talk about the design and implementation issues of GNA95GP. In the finally Chapter, we summary our work.

OKLAHOMA STATE UNIVERSITY

2. Literature Review

2.1. A Review of Standard Graphics Packages.

In [17], a graphics package is defined as a software system. A graphics package is built from a set of subroutines or functions. Application programs use these subprograms to generate images on an interactive display device and to receive data from input devices. A major advantage of a graphics package is that it makes it possible for an application programmer to concentrate on his (her) application without being concerned with low-level functions related to interactive graphics programming. So, application system development time can be reduced. Also, programmers who are less familiar with interactive graphics programming will find a graphics package useful in developing applications. With the increasing emphasis on the use of computer graphics in Computer Science and other application areas, more and more graphics packages ranging from simple raster graphics packages to advanced 3-D sophisticated graphics packages are provided to different levels of users. These packages are developed for different platforms and different programming languages [2]. Several packages are identified below:

2.1.1. GKS (Graphical Kernel System).

GKS is an American National Standards Institute (ANSI) standard graphics library, and a superset of the International Standardization Organization (ISO) standard graphics library. GKS provides subroutines for an application programmer to use within a program in order to produce and manipulate graphical images. GKS is a language

OKLAHOMA STATE UNIVERSITY

independent standard. Currently, GKS has bindings to several languages including FORTRAN, Pascal, C, Alsys Ada and Verdix Ada [8,9,11]. The layer model represented in Figure 1 illustrates how GKS fits within a graphical system. Each layer may call the functions of the adjoining lower layers. An application program will have access to a number of application-oriented layers, the language-dependent GKS layer, and the operating system resources. The top interface of the GKS nucleus is the language-independent application interface and is defined by the GKS standard. The interface between the language-dependent layer and the application layer is the language-dependent application interface, e.g., FORTRAN or Pascal interface. GKS serves as the nucleus that provides basic graphics capabilities for many different applications. It also defines the graphical functions without reference to special graphical devices. In GKS, a graphical output device and the input devices connected to it are called a graphical workstation. Device drivers translate device-independent representations of functions within the GKS nucleus to and from the different graphical devices [15]

OKLAHOMA STATE UNIVERSITY

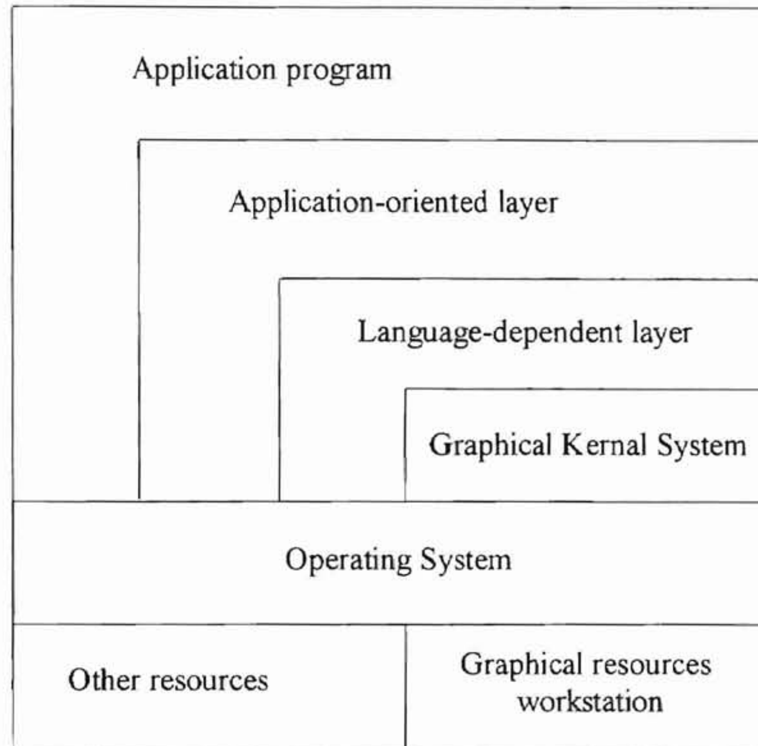


Figure 1. Layer model representation in GKS.
(adopted from [15])

From references [20,21,22], we summarize the main features of GKS below:

a) Logical Workstations

GKS introduces the concept of a logical workstation. In general, workstations are abstractions of graphics devices. A single display surface and its associated set of input peripherals attached to a computer are the components of a workstation in GKS [23] Figure 2 demonstrates the terminology of a workstation.

OKLAHOMA STATE UNIVERSITY

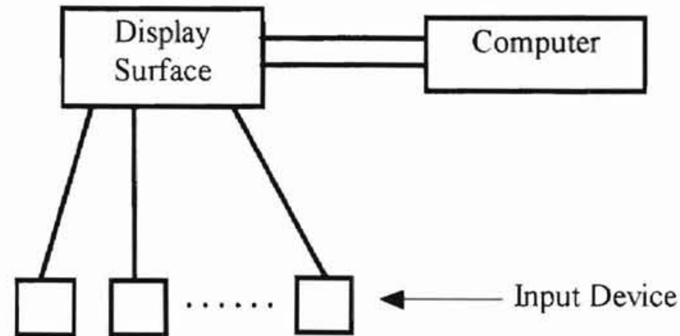


Figure 2. Description of a workstation.
adopted from [23]

b) Graphics Primitives

GKS provides six primitives, they are listed below:

POLYLINE -- GKS generates a polyline with given corner points.

POLYMARKER -- GKS generates at each point of a given set of points a centered symbol.

TEXT -- GKS generates a character string at a given position.

FILL AREA -- GKS generates an area defined by a given set of points. The area may be filled with a uniform color, a pattern, or a hatch or maybe identified by only drawing boundary.

CELL ARRAY -- GKS generates a raster picture out of a given cell array.

GENERALIZED DRAWING PRIMITIVES -- GKS deals with special geometrical output capabilities of a workstation. These output capabilities include drawing spline curves, circular arcs, and elliptic arcs. The objects are characterized by an identifier, a set of points and additional data. GKS applies all transformations to the points but leaves the interpretation to the workstation.

OKLAHOMA STATE UNIVERSITY

c) logical Input Devices

GKS classifies input devices into the six categories:

locator -- specifies a position by its x and y coordinates.

pick -- identifies a display object.

choice -- selects from a set of alternatives.

valuator -- inputs a value.

strings -- inputs a string of character.

stroke -- inputs a sequence of (x,y) positions.

d) Mode of Interaction

All logical devices can be used via three modes: REQUEST, SAMPLE, and EVENT. In REQUEST mode, a device is read only when some user action occurs. In SAMPLE mode, a device is read immediately without waiting for any user action. In EVENT mode, if a device gets input from user, it will insert the input in a queue. There is only one queue for all classes of logical devices.

e) Segmentation

In GKS, Graphics primitives can be grouped together in segments, this grouping of the graphics primitives of a picture into a hierarchy is called segmentation. The elements in a segment can be manipulated together as a unit. In other words, a segment is a set of graphics primitives that are arranged together in an appropriate data structure. The programming language and the design of the graphics system determines what types of data structures can be used to realize a segment. Each segment is characterized by a name

OKLAHOMA STATE UNIVERSITY

which is defined by the application. The operations that can be applied on a segment are *open, close, make visible, make invisible, transform (scale, translate, rotate), copy, select, rename, highlight, insertion,* and *delete.*

f) Metafile

A metafile is a means to transmit and store pictures in an application-independent and device-independent way. A file that includes information that is in device-independent display-record format for generating images is called a metafile. Metafiles are used to store graphics images. GKS provides functions to read and write a metafile. A GKS metafile is treated as an output workstation. Each metafile is a sequence of items, each of which has the following components:

a. Item type.

b. Item data record length.

c. Item data record.

2.1.2. PHIGS (Programmer's Hierarchical Interactive Graphics System).

The design objective in PHIGS is to support computer graphics applications that are highly dynamic and interactive. PHIGS includes a hierarchical graphical database. The significance of the database lies in its capability to be edited while elements of the database are being displayed. Such functionality is needed to support applications such as computer aided design/computer aided manufacturing (CAD/CAM) systems, command_control systems, modeling of objects, and so on. It also supports multiple platforms and has a

OKLAHOMA STATE UNIVERSITY

variety of programming language bindings. Alsys Ada, Verdix Ada and Meridian Ada compilers support bindings to PHIGS [9,10,11].

PHIGS is a defined by ANSI (American National Standards Institution). According to references [20,21,22], we summarize the following concepts and features which PHIGS supports.

a) Graphics Output

The graphics output facilities of PHIGS are identical to those of GKS-3D. It offers the following 3D-primitives:

POLYLINE

POLYMARKER

TEXT

FILL AREA -- An area defined by a given set of points, the area may be filled with a uniform color, a pattern, or a hatch.

FILL AREA SET -- PHIGS generates an area defined by a given set of points. Areas with holes can thus be defined. The fill area set has its own attribute set which allows the indication of the display of the area's interior (uniform color, pattern, hatch, or empty) and the adjustment of the display of the boundaries (line type, line width, and color).

CELL ARRAY -- PHIGS generates a raster picture from a given pixel array, the given pixel array is defined as a CELL ARRAY.

GENERALIZED DRAWING PRIMITIVE (GDP) -- This primitive is the same as the GDP described in GKS.

OKLAHOMA STATE UNIVERSITY

b) Graphics Input

PHIGS defines six different input classes that correspond to the six possible input data types. They are LOCATOR, STROKE, VALUATOR, CHOICE, PICK, and STRING, they are the same as those defined in GKS. Each logical input device can be operated in three different modes.

c) Interaction handling

Like interaction handing in GKS, in PHIGS, each logical input device can be operated in three different modes. They are REQUEST, SAMPLE and EVENT. Input values are entered differently and are passed to the application program depending on the mode.

REQUEST -- A call to the REQUEST function tries to read a logical input value from the logical input device specified, PHIGS waits until the operator has entered the input or has caused an interrupt by pressing, for instance, a special key.

SAMPLE -- On calling the SAMPLE function, the current logical input value of the logical input device specified is reported without waiting for an operator action.

EVENT -- PHIGS maintains an input queue containing chronologically ordered event reports. An event report contains the identification of the logical input device and the logical input value.

d) Storage

Creation and manipulation of individual parts of a picture are convenient features for all graphics applications. PHIGS supports this by using the concept of a *structure*. A

OKLAHOMA STATE UNIVERSITY

structure is a labeled group of primitives. *Structures* may be edited. They can also have substructures. *Structures* are organized as a graph called *structure network*. Figure 3 shows an object and a corresponding model as a *structure network*. *Structures* are stored in a database called *Central Structure Storage* (CSS).

As *structures* are created, they are added to the CSS. The root of the graph represents the whole object (or picture). *Structures* are displayed by "posting" them to output devices.

While, In GKS, picture definition and picture output are combined, PHIGS distinguishes between picture definition, picture editing, and picture representation. Figure 4 shows PHIGS organization model.

OKLAHOMA STATE UNIVERSITY

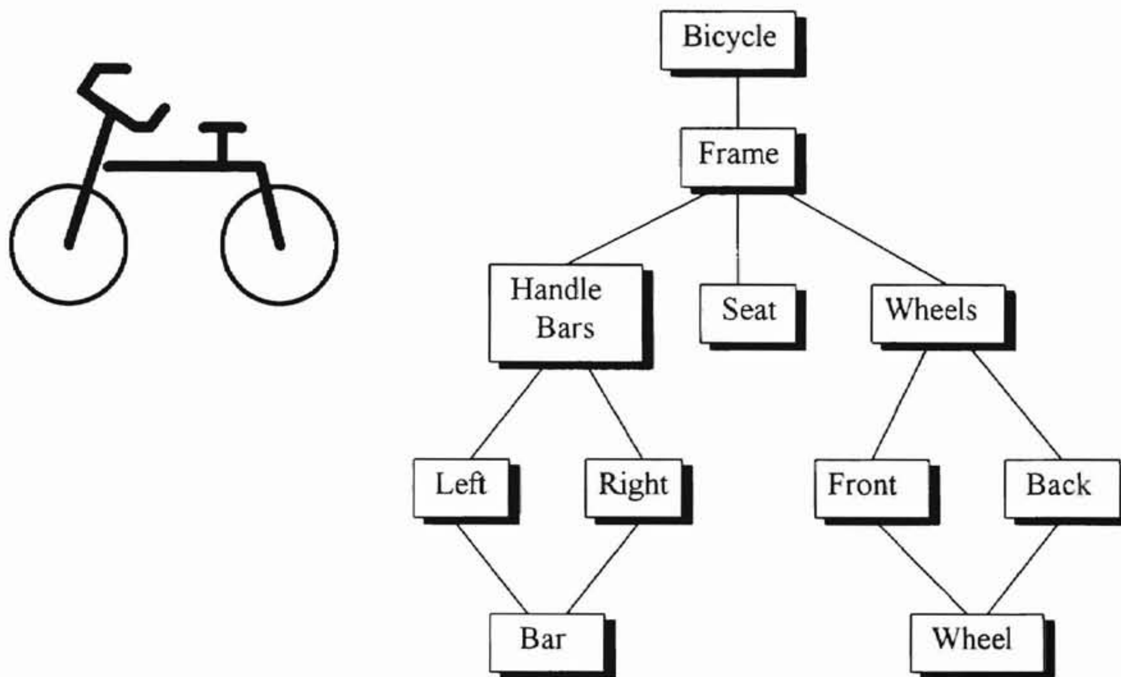


Figure 3. Example of a graphics object storing in PHIGS-CSS.
(adopted from [20])

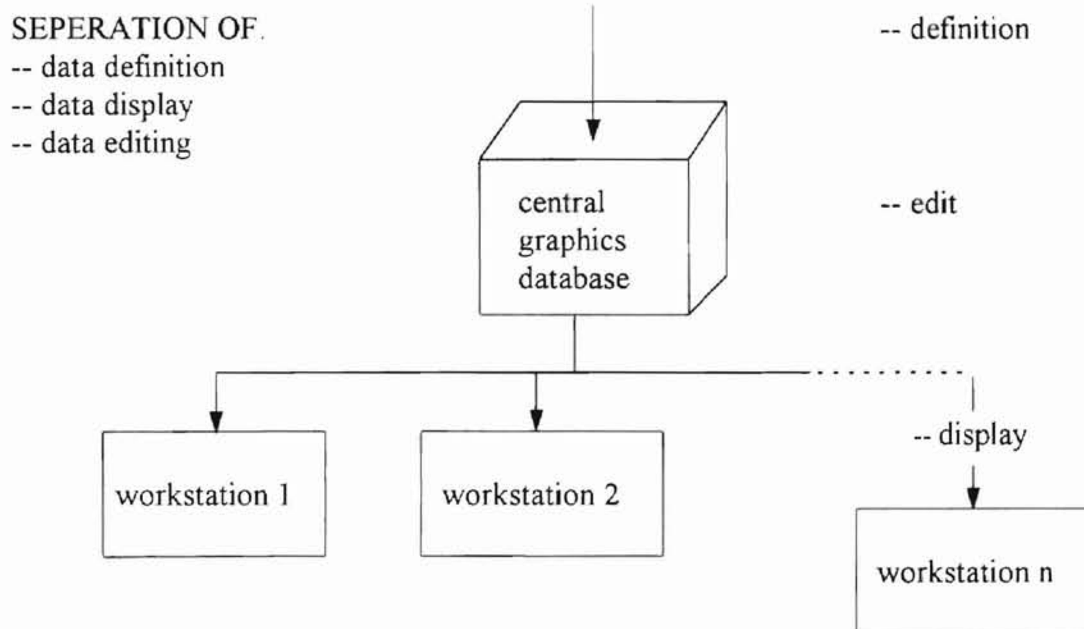


Figure 4. PHIGS organization model (adopted from [20]).

OKLAHOMA STATE UNIVERSITY

2.1.3. SRGP (Simple Raster Graphics Package).

SRGP is a device-independent software. It is designed for raster graphics and it exploits the capabilities of raster graphics. The output primitives (lines, rectangles, circle and ellipses, and text strings) supported in SRGP are similar to that of the popular Macintosh QuickDraw raster package [2]. The Xlib package of the X-Windows System graphics package also provides similar primitives [2]. SRGP's interaction-handling features are a subset of those of SPHIGS. SRGP differs from QuickDraw and Xlib in the interaction handling feature. SRGP is implemented on X Windows platform and on PCs. The programming languages used are PASCAL and C [2]. A detailed description of SRGP can be found in [2]. We summarize the following main features of SRGP here.

a) Drawing Primitives

SRGP supports a basic collection of primitives: lines, markers, polygons, circles, ellipse, and text. It also supports filled primitives, such as filled_polygon, filled_circle, filled_ellipse.

b) Drawing Attributes

SRGP provides a set of attribute setting routines for drawing primitives, they are used to set *line style, line width, marker style, marker size, text font, filling style, filling bitmap pattern, front color,* and *background color.*

c) Interaction Handling

There are two basic techniques which are used to receive the device input created by a user. They are sample and event modes. In *sample* (also called *polling*) mode, the

OKLAHOMA STATE UNIVERSITY

application program queries the current value of a logical input device and continues execution. A user performs a sampling to determine whether or not the device's state or value has changed since the last sampling. Only by continuous sampling of the device, can a user application know the changes in a device's state. This mode is very costly for an interactive application, since an application spends most of its CPU cycles in the sampling loop waiting for measure changes. An alternative approach is the *interrupt-driven* interaction (*event* mode). In this technique, the application enables one or more devices for input and then continues execution, in the background SRGP monitors the devices and stores information about each *event* in an event queue (a change in a device's state caused by the user action is called an *event*). At its convenience, the application checks the event queue and processes the events in the event queue in temporal order. When an application checks the event queue, it specifies whether or not to enter a wait state. If the queue is not empty, the event at the head of the queue is removed for application to process. If the queue is empty and a wait state is not specified by the application, the application is free to execute. If the queue is empty and a wait state is specified by the application, the application is blocked and it waits until the next event occurs, the application can also specify the maximum waiting time interval for itself to wait for until it is free to continue with execution in this case. In effect, comparing with sampling of the input, event mode is much more efficient. Figure 5 illustrates conceptual interaction handling in SRGP.

OKLAHOMA STATE UNIVERSITY

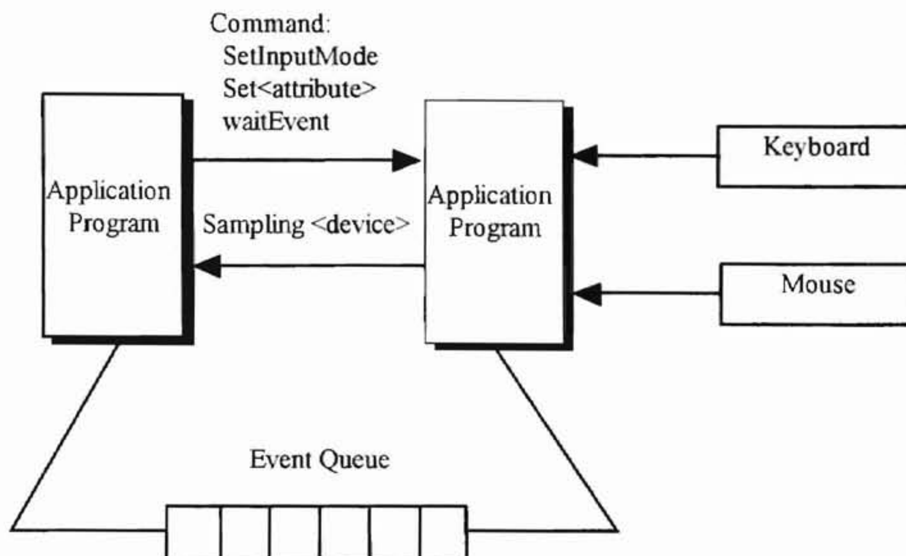


Figure 5. Sampling verus event-handing using the event queue (adopted from [24]).

d) Limitations of SRGP

Two limitations of SRGP are discussed in [21]. One limitation concerns the coordinate system of SRGP, since for those applications that require great precision and range, the machine-dependent integer coordinate system of SRGP is too inflexible. Another limitation of SRGP is the storage scheme used. SRGP stores an image in a canvas as a matrix of unconnected pixel values rather than as a collection of graphics objects (primitives), so it does not support object-level operations, such as *delete, move*, and *change color*.

2.2. Review of Ada 95.

Data abstraction is a powerful concept in programming. Data abstraction combines a data type with available operations on this data type. The philosophy is: we can use such

data types without knowing the details of its representation. A program using services of another program is called a client program. A client program can declare and apply the available operations on the objects of a data type without knowing the structure of this data type and implementation of this data type's operators, these details are hidden from the client programs. So, it separates the *use* of data and operators (by a client program) from the *representation* of data and *implementation* of the operators (within the abstract data type). Later, we can change the internal implementation without affecting the client program. Ada 95 provides many capabilities to develop ADTs. The following is a summary of these abstraction features from [1,3,4,18].

Packages

A package is a language mechanism to bundle objects together. Types, subprograms, constants and so on can be bundled into a single module as a package and be made available to a client. A package specification serves as a *contract* between the implementor of the package and the user (client program). All types and functions "promised" in the specification must be implemented in the package body (invisible to the user), and the client programs must use the package source correctly [18].

Private types

Limited private types and *private types* support abstraction and information hiding because the structure of the data type is hidden from the client program. These two private types allow the implementation to change without affecting the rest of the program.

OKLAHOMA STATE UNIVERSITY

Operator overloading

Operator overloading allows us to write new arithmetic and comparison operators for new types and use them just as we use the predefined operators.

User-defined *exception*

User-defined exception allows the writer of a package to provide the exceptions to client programs in order to signal to a client program when an exception occurs within a function or procedure supplied to the client in the package.

Genericity

Genericity is powerful in software reuse. It allows us to write subprograms and packages that are so general that they do not even have to know all the details of the types they manipulate; these types can be passed to generic unit as parameters when a generic unit is *instantiated*.

Encapsulation

Encapsulation is supported by Ada's package and private type features. A package forms a collection of logically related entities or resources, and a package encapsulates these resources. The specification of package exports these entities. The entities include two categories; one is objects and types, the other is operations. The body of a package is hidden from client programs. Private types *encapsulate* all the type details from the client program.

OKLAHOMA STATE UNIVERSITY

Inheritance

Inheritance means a new type can take on some or all the properties of an existing type. Ada provides derived types, and also a more powerful construct called a tagged type. A tagged type can inherit all the properties of an old type and also can be extended in the future without a programmer changing the original type declaration.

Polymorphism

In [18], Professor Feldman defines the concept of polymorphism as follows:

> "*Polymorphism* is the ability of methods (or functions, or procedures) with the same name to exhibit different behaviors depending on the type (or class) of objects that receives when the methods is called."

Ada supports polymorphism by procedure and function overloading, and extends it significantly through the concept of *dynamic dispatching*.

OKLAHOMA STATE UNIVERSITY

3. GNAT 95 Graphics Package Design and Implement Issues.

3.1. Software Dependency in GNA95GP.

GNA95GP is developed based on three free software systems; they are DJGPP, GNAT Ada 95, and RSXWDK. DJGPP [12] provides 32-bit GNU-C program development system for personal computers. GNAT [13] is an Ada 95 complier which is integrated into DJGPP. RSXWDK [12] is a windows software development kit; it allows programmers to develop GNU-C programs to call windows functions. Figure 6 illustrates the dependency relationship among a user program, GNA95GP and the above three software systems. It gives a clear abstract system view.

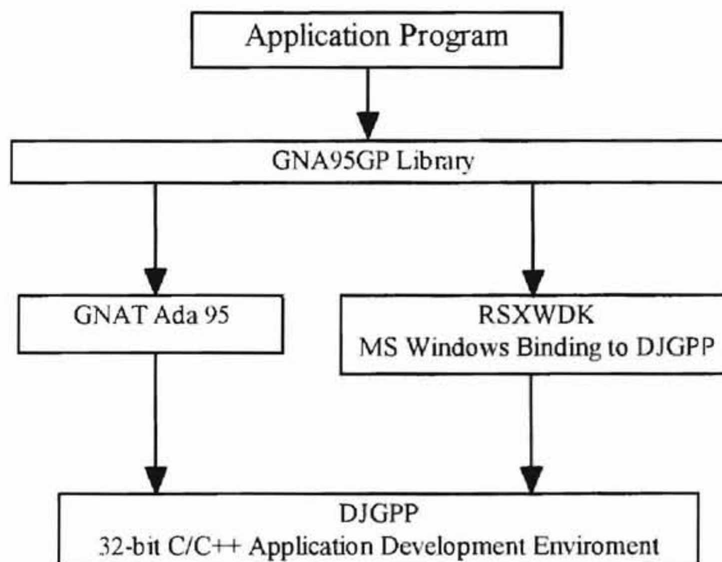


Figure 6. Software dependence in GNA95GP.

OKLAHOMA STATE UNIVERSITY

3.2. General Approach to GNA95GP Design.

From the top view, GNA95GP is a set of Ada 95 packages to let users write Ada 95 graphics applications which can be run under the MS_Windows. In the design of GNA95GP, we use structured and object-oriented design methods. GNA95GP consists of three major components. Figure 7 shows those components in GNA95GP. These three components are graphics_primitives (2D primitives), interaction handling, and storage structure. Graphics_ primitives packages provide basic drawing routines for users to generate graphics image in the screen. Interaction handling packages support monitoring dynamic input messages from the user and handling them. Storage structure package lets users store their graphics objects drawn in the screen. These three components are integrated to support several features of graphics standards in GNA95GP. In the design of graphics primitives, we followed some principles of SRGP to achieve the goal of ease of use. The storage scheme in GNA95GP is inspired by the structure storage model of PHIGS. A restricted model is used in GNA95GP.

In the remaining sections of this chapter, the discussion concentrates in design and implementation details of graphics_primitives and storage structure.

OKLAHOMA STATE UNIVERSITY

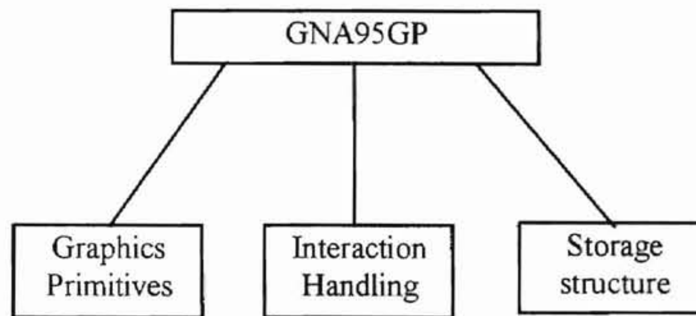


Figure 7. Components inside GNA95GP.

3.3. Design and Implementation of Graphics primitives.

3.3.1. Primitives and Attributes.

The Ada Packages of graphics primitives provide basic routines for output graphics primitives and their associated appearance attributes. Table 1 lists the routines for graphics primitives along with their descriptions. Table 2 describes the routines for appearance attributes and their descriptions.

3.3.2. Implementation of graphics primitives.

There are a set of packages that serve as vehicles of design and implementation of graphics primitives. Figure 8 shows the design approach to graphics primitives by using Booch diagram. These packages' names and functionalities are listed and described in table 3. Each package encapsulates a concept, for example, output primitives are bundled in the output package.

To serve as an example, appendix_I is an application program which uses primitive packages in GNA95GP, and its image output is shown in Figure 9.

OKLAHOMA STATE UNIVERSITY

Table 1. Output primitives and descriptions.

Routine Name	Description
Gna95gp_DrawPixel	draw a pixel at a point (x,y).
Gna95gp_line	draw a line from point (x1,y1) to point (x2,y2).
Gna95gp_Rectangle	draw a rectangle with top-left corner at (x1,y1) and bottom-right corner at (x2,y2).
Gna95gp_PolyLine	draw a polyline defined by a given set of points.
Gna95gp_ellipseArc	draw a elliptic arc given by its extent rectangle, starting point (x1,y1) and ending point (x2,y2).
Gna95gp_DrawMarker	draw a marker with specified shape and size.
Gna95gp_text	output a text string beginning at position (x,y).
Gna95gp_FilledRectangle	draw a rectangle with a given filling pattern and color.
Gna95gp_Polygon	draw a polygon with a given filling pattern and color.
Gna95gp_FilledEllipse	draw a ellipse with a given filling pattern and color.

Table 2. Attribute routines and descriptions.

Routine name	Descriptions
gna95gp_setLineStyle	set line style to be one of CONTINUES, DOTTED, DASHED, and DOTTED-DASHED.
gna95gp_setLineWidth	set line width.
gna95gp_setColor	set front color and text color.
gna95gp_setBackgroundColor	set background color.
gna95gp_setFillStyle	set filling style to be one of (SOLID, BITMAP_PATTERN_TRANSPARENT, BITMAP_PATTERN_OPAQUE).
gna95gp_setFillBitmapPattern	set filling bitmap pattern by specifying the pattern index.
procedure gna95gp_seWriteMode	set write mode to be one of (REPLACED, OR, XOR, AND)

OKLAHOMA STATE UNIVERSITY

TABLE 3-- Package names

Package Name	Package Description
Output package (output)	Implements drawing primitives
Init Package (init)	Contains system initialization functions
Attribute Package (attribute)	Functions for setting attributes
Attribute Type Package (attrtype)	Data type declaration for attributes
Canvas Management Package (canvata)	Canvas management functions
Canvas Type Package (canvtype)	Type declarations for canvas management
Windows type Package (wintype)	Windows handler type declaration
Color Package (color)	Color type declaration and operations
Bitmap Pattern Package (bitmappa)	Type declarations for bitmap pattern
Object Type Package (objtype)	Drawing object type declarations (e.g. point)

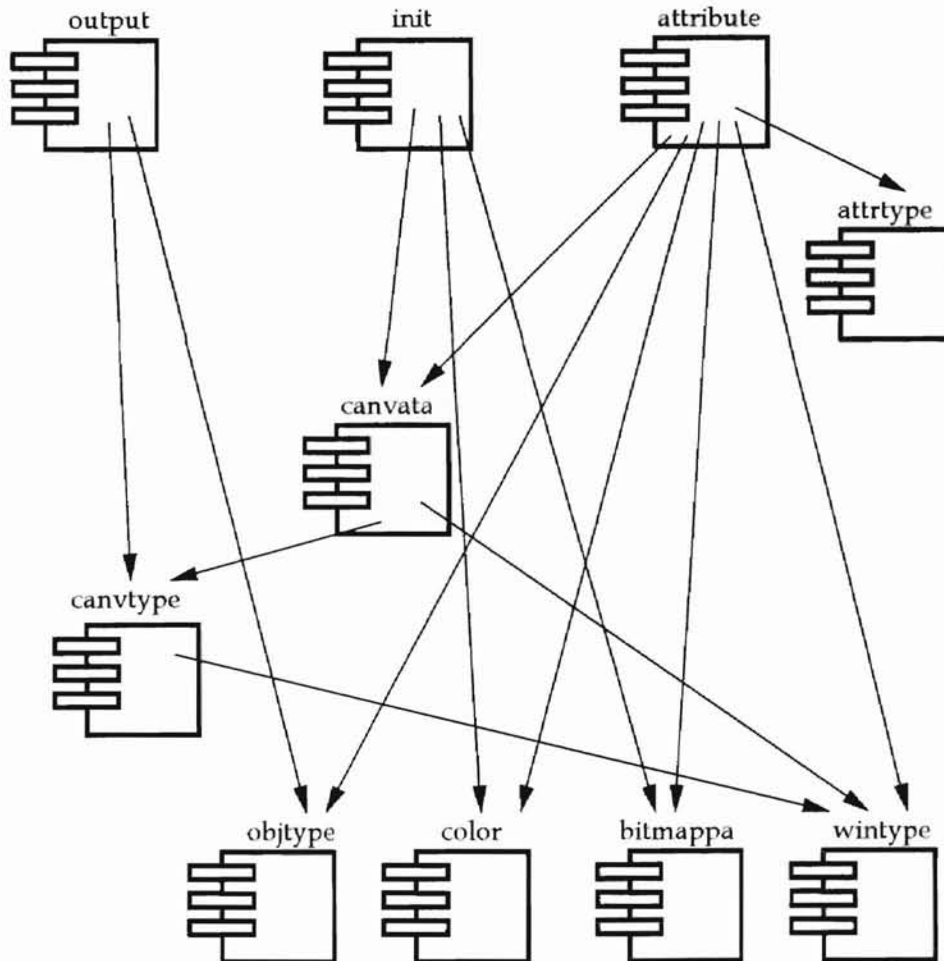


Figure 8. Design of drawing-packages using Booch diagram.

OKLAHOMA STATE UNIVERSITY

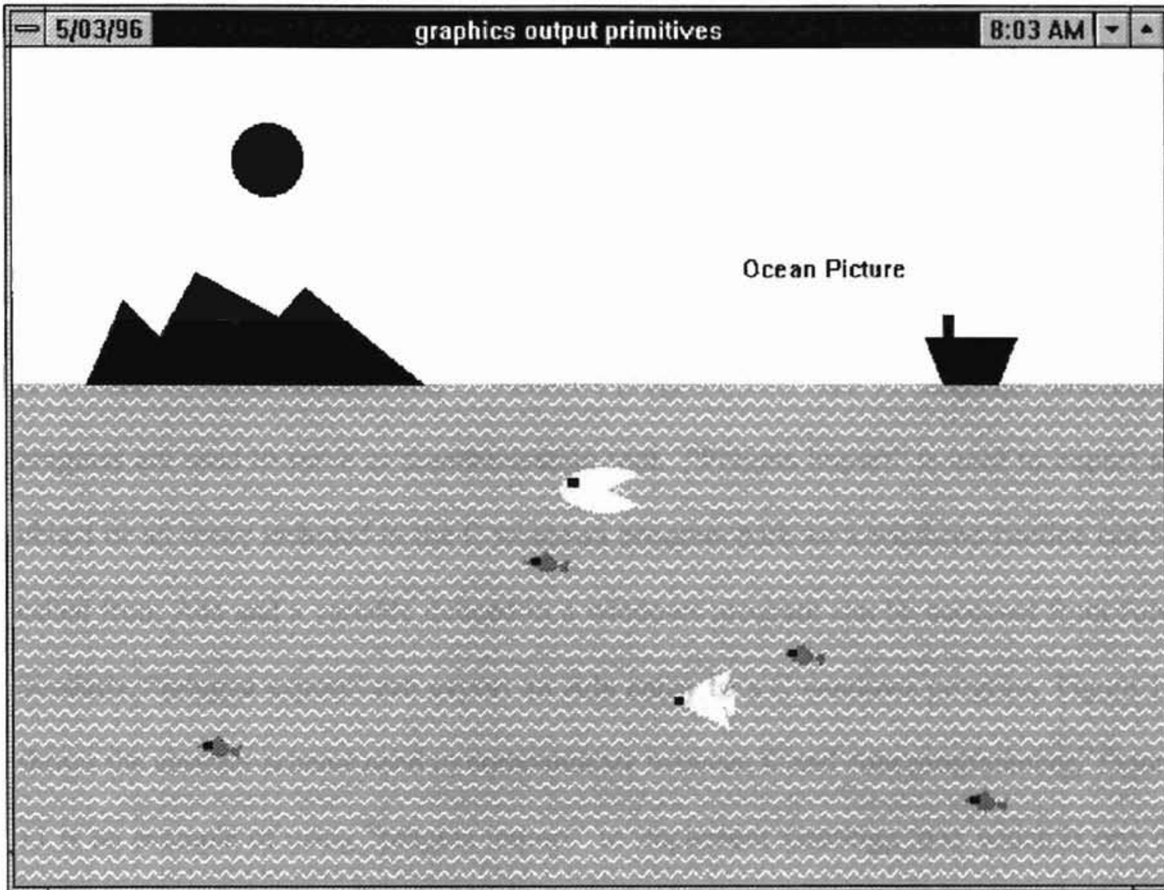


Figure 9. Sample output.

OKLAHOMA STATE UNIVERSITY

As we mentioned before, GNA95GP is developed based on GNAT Ada 95 and RSXWDK. It is true that we provide GNAT Ada 95 packages of graphics primitives for users to program their graphics applications, and these package bodies are also implemented in Ada 95. However, in the low-level implementation of these graphics primitives, we employ windows graphics drawing functions to do the real work. In order to let Ada 95 communicate with C windows program, two Ada 95 language interfacing pragmas are used to serve as communication tool. "Pragma import" is used to import an object or an entity included in the C windows program part, so a windows routine can be called from Ada and a variable defined in a windows program can be accessed from Ada. Similarly, "pragma export" can export an Ada entity to a C windows program. Figure 10 shows the abstract scheme used to communicate between C and Ada programming. Figure 11 is an example of using "pragma import" to implement Gna95gp_text which is one of graphics primitives used to output a text string at a given position. The C language procedure WIN_text is a windows C program which draws the text. This implementation method is used for all graphics primitives

OKLAHOMA STATE UNIVERSITY

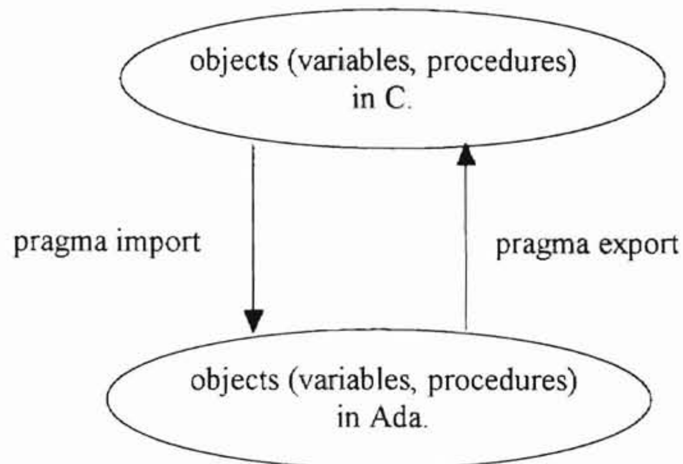


Figure10. Communication scheme used between C and Ada 95

```
package body Output is
  ...
  -----------------------------------------------------------------------------
  procedure WIN_Text(handle: HANDLE; x:integer; y:integer; text:
                                      String; textSize:integer);
  pragma import (C,WIN_Text, "Win_Text");
  procedure Gna95gp_text(point: in tagPoint; text: String) is
  x, y:        integer;
  size:        integer;
  Handle:      drawType;
  begin
    x := Xof(point);
    y := Yof(point);
    Handle:= getWinHandle;
    size := text'Last - text'First + 1,
    WIN_Text(Handle, x, y, text, size);
  end Gna95gp_text;
  -----------------------------------------------------------------------------
  :
  :
  :
end Output;
```

Figure 11. Example -- Implementation of a primitive.

OKLAHOMA STATE UNIVERSITY

3.4. Design of interactive handling.

In many cases, user input actions involve in graphics application dynamically. In order to trace these user input messages and handle them properly, like the interaction handling in SRGP, GNA95GP provides two types of interaction handling. They are measure_mode and event_mode handling. Like SRGP, in *measure_mode* handling, the application program queries the current state or value of an input device. If an application wants to know any changes in input state, it has to spend most of its CPU cycle to do busy and tight measuring loop. On the other hand, in *event_mode*, the application enables mouse or keyboard or both of the devices for receiving input in its event queue. The application removes these messages from the queue to process them at its available time. A mouse button pressed down or a keyboard key pressed is considered to be an input event, and any input event occurrence is monitored by GNA95GP and GNA95GP places it in the event queue. This interaction handling model is the same as the one shown in Figure 5. Figure 12 describes the design to implement this interaction model. GetMessage or PeekMessage function is called to retrieve or peek an user input from the message queue managed by the windows system. This user input message is dispatched to this window by calling DispatchMessage function. We can see how program control goes from user program to device handler "WndProc" which handles the user input message received from all kinds of input devices. For details of the implementation of interation handling, please refer to Kuang's thesis [26].

OKLAHOMA STATE UNIVERSITY

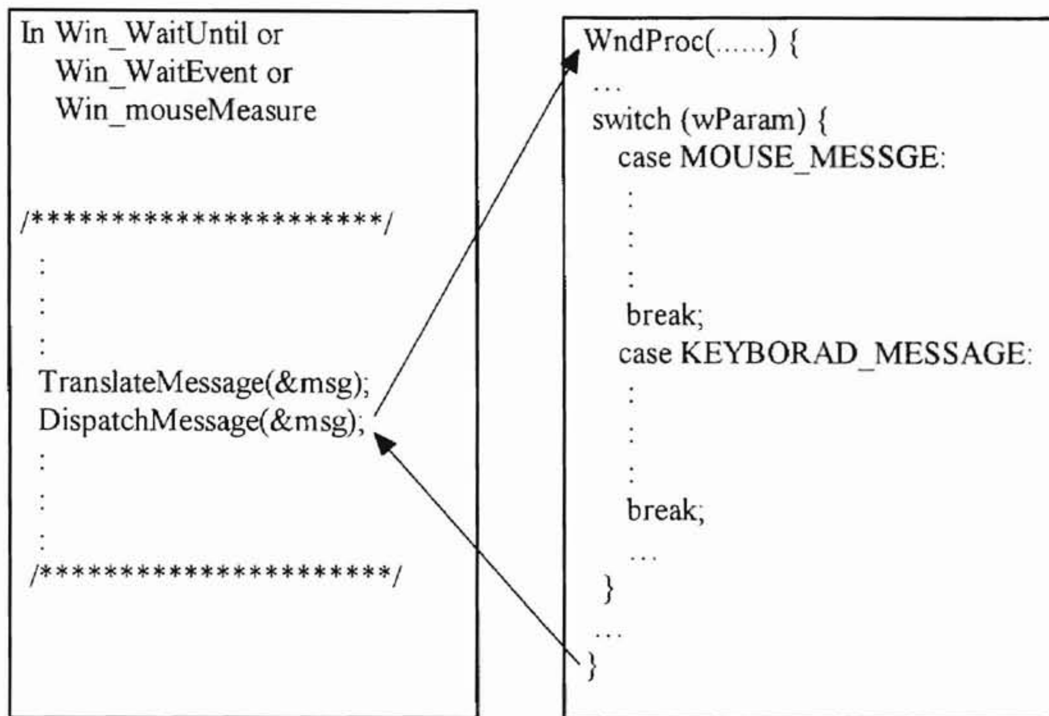


Figure 12. Interaction handling model in GNA95GP.

3.5. Storage of graphics primitives.

For many graphics applications, especially those with animation features, graphics objects drawn in the screen need to be dynamically modified. For this purpose, graphics package should provide some kind of mechanism to store the graphics objects existing in the screen in order to modify them later. The design of storage scheme in GNA95GP is adopted from the central structure storage (CSS) scheme of PHIGS. The main difference of storage scheme between PHIGS and GNA95GP is that PHIGS supports *substructure*, that means PHIGS storage scheme supports structure hierarchy. We will introduce the concept of *structure* and explain the details of structure hierarchy in the following section.

OKLAHOMA STATE UNIVERSITY

3.5.1. Design of storage scheme.

Like the storage scheme in PHIGS, GNA95GP maintains a database of structures. A **structure** in GNA95GP is a sequence of **elements**, the **elements** include primitives_generating functions and appearance attributes_setting functions. A **structure** is used to describe a graphics object which is grouped by a set of graphics primitives and their appearance attributes. Each structure has a unique ID. Structures are objects that can be edited. The manipulations permitted on a **structure** are "open", "close", and "delete". "Open" operation initiates editing of a **structure**. "Close" operation terminates the editing of a **structure** initiated by "open". "Delete" operation deletes a existing **structure**. The following three routines are used to perform these three operations.

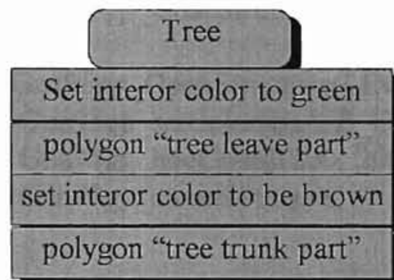
```
procedure Gna95gp_openStructure(ID: integer);
procedure Gna95gp_closeStructure;
procedure Gna95gp_deleteStructure(ID: integer);
```

Figure 13 is an example of a structure, Figure 13-a shows the code used to generate this structure whose ID is TREE_STRUCT, (b) Figure 13-b describes the components of the structure, and Figure13-c shows this graphics object as an image on the screen.

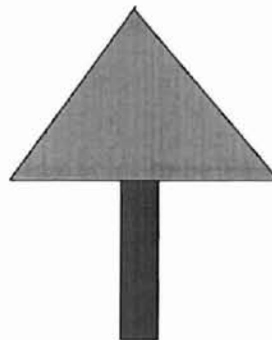
OKLAHOMA STATE UNIVERSITY

```
Gna95gp_openStructure(TREE_STRUCT);
Gna95gp_setColor(GREEN_COLOR);        --set leave to be green.
Gna95gp_polygon(3, vertics_1);        --for the shape of leave part.
Gna95gp_setColor(BROWN_COLOR);        --set tree trunk to be brown.
Gnagp_polygon(4, vertics_2);          --for the shape of tree trunk.
Gna95gp_closeStructure;
```

(a) Code segment to generate structure of TREE_STRUC.



(b). Tree structure.



(c).Image of tree object.

Figure 13. An example structure.

The attributes inheritance rule used in SRGP is that the appearance attributes state remains the same until it is changed explicitly. This rule frees programmers from specifying a long list of attribute parameters for each primitive. In GNA95GP, this rule is modified. The attributes state set by attribute elements only applies to the primitives inside the current *structure*. The former attribute state inside the previously opened *structure* doesn't apply to the primitives in current *structure*. Figure 14 gives a simple example to demonstrate how the attributes' state apply to primitives in GNA95GP. In editing LINE_STRUCT_1 *structure*, the color attribute state is set to be blue and it applies to the

OKLAHOMA STATE UNIVERSITY

primitive inside this *structure* (the line). After closing the *structure* of LINE_STRUCTURE_1, the color attribute state is not BLUE, it becoming unpredictable. That is why in editing LINE_STRUCT_2, color attribute state need to be set again in order to store a blue line in LINE_STRUCT_2 *structure*.

```
...
Gna95gp_OpenStructure(LINE_STRUCT_1);
Gna95gp_setColor(BLUE_COLOR);            --set color to be BLUE.
Gna95gp_lineCoord(16, 16, 160, 160);     --draw a line.
Gan95gp_closeStructure;

Gna95gp_OpenStructure(LINE_STRUCT_2);
Gna95gp_setColor(BLUE_COLOR);            --set color to be BLUE.
Gna95gp_lineCoord(200, 200, 250, 250);   --draw a line.
Gan95gp_closeStructure;
...
```

Figure 14. An example of attribute rule applying to primitives.

The elements inside a structure are indexed from 1 to N. When an element is inserted or deleted, the index associated with each high-indexed element in the same structure is increased or decreased by 1. The **current element** is an element whose index is stored in the **current_index** variable. When a structure is opened with "Gna95gp_openStructure", the **current_index** is set to N, where N is the index of last element inside this structure (it is equal to 0 when a structure is opened for the first time) The **current_index** is increased by 1 when a new element is inserted after the current element. The **current_index** is decreased by 1 when the current element is deleted. The operations permitted on the **current_index** are listed below:

OKLAHOMA STATE UNIVERSITY

procedure Gna95gp_setElementIndex(index: integer);

procedure Gna95gp_OffsetElementIndex(offset: integer);

-- if offset >0, movement is forward, and if offset < 0, movement is backward.

Insertion and deletion are two common types of operations permitted inside a structure. When a new element is inserted immediately after the current element, the current index is moved to point to the new element. The following functions are used to apply delete operation.

procedure Gna95gp_deleteElement(); -- delete current element.

procedure Gna95gp_deleteElementInRange(firstIndex: integer, secondIndex: integer);

After a deletion, The **current_index** moves to the element immediately preceding the element which is deleted , in the range deletion case, it moves to the element immediately preceding the first element which is deleted. After a deletion, all the indices of the elements following the element deleted are recalculated. Figure 15 gives the code segment used to modify TREE_STRUCTURE in figure 13. Figure 16 shows a snapshot of TREE_STRUCT *structure* as editing progresses.

OKLAHOMA STATE UNIVERSITY

```
Gna95gp_openStructure(TREE_STRUCTURE);
Gna95gp_setElementIndex(1);
-- delete element which sets tree color to be green
Gna95gp_deleteElement;
-- insert a element which sets leave part to be red
Gna95gp_setColor(RED_COLOR);
-- move to element which generates shape of the tree.
Gna95gp_offsetElementIndex(1);
-- delete the element which generates the shape of the tree.
Gna95gp_deleteElement;
-- insert a element which generates the shape of leave part.
Gna95gp_polygon(5, vertics_3);
```

Figure 15. Code segment used to edit TREE_STRUCTURE.

OKLAHOMA STATE UNIVERSITY

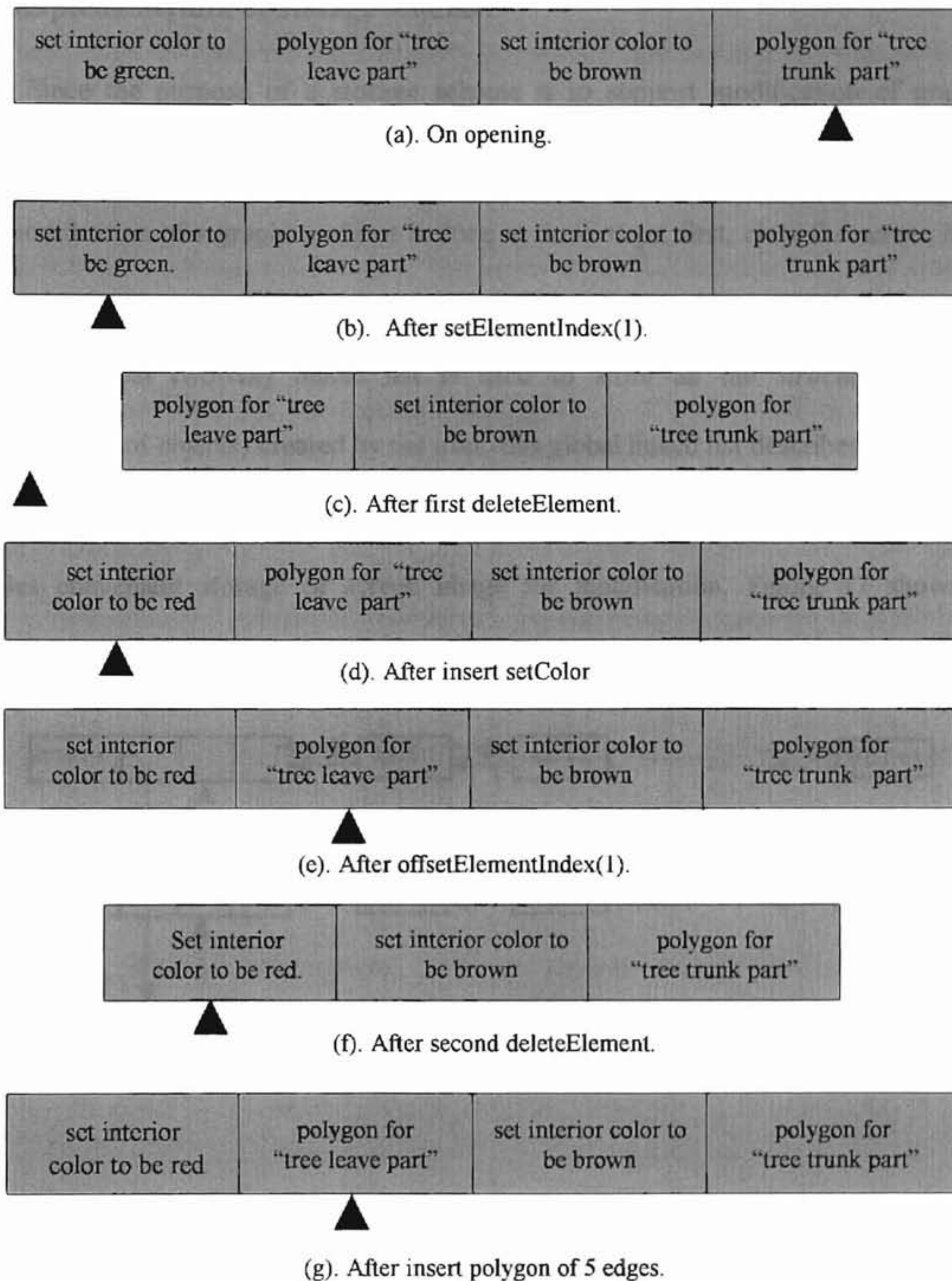


Figure 16. Sequence of TREE_STRUCTURE during editing.

3.5.2. Implementation of storage scheme.

Since the purpose of a storage scheme is to support modification of graphics objects shown in the screen, each object need to be stored in a proper data item. In this way, modification of a graphics object is done in three steps, first, clear the screen image of this object, then modify this data item, finally redisplay this updated object in the screen. A global two-way linked list is used to store all the *structures* (internal representations of objects) created by the user, this global linked list describes the image in the screen, it takes up less space and is more device-independent than bitmap method, it provides convenient storage of screen image for modification. Figure 17 shows the structure of this linked list.

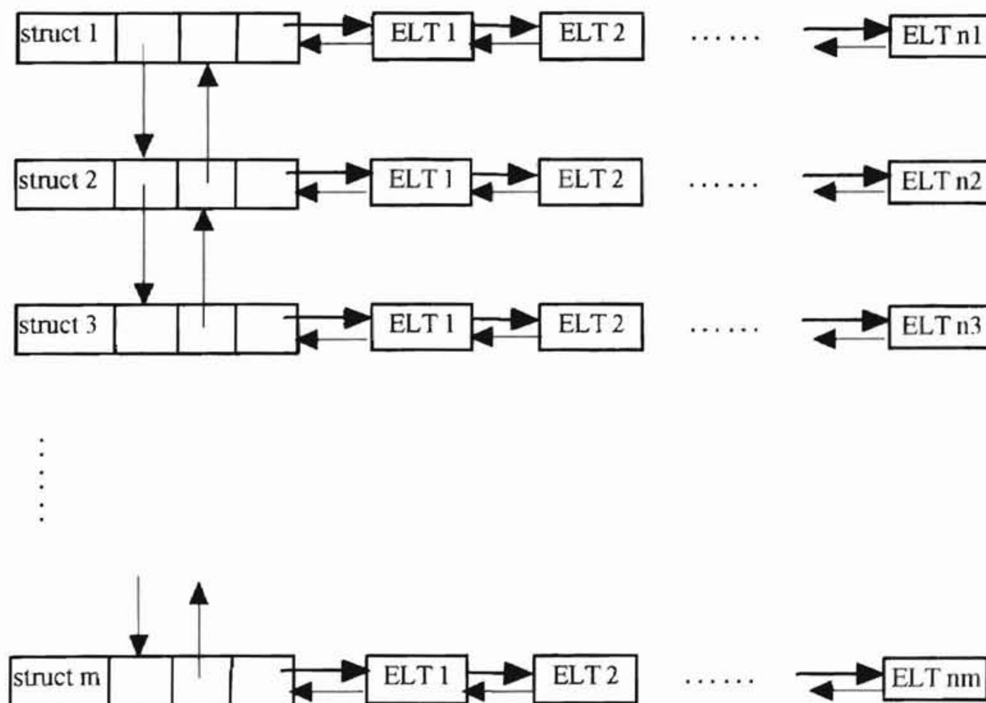


Figure 17. Global linked list for implementation of storage scheme.

Obviously, the reason for using two-way linked list instead of a one way linked list to implement is that the former is more flexible for insertion and deletion of nodes in the list, since high speed of displaying object image in graphics application is very important. At the beginning of the user application, this linked list is empty since no graphics object is created. A node is dynamically allocated and added to the link list when the user creates a structure which is associated with a graphics object. In this global list, there are two types of nodes, they are *structure_node* and *element_node*. A *structure_node* is used to store the information of a structure, such as ID of the structure, number of elements inside this structure, and entry to the first element. An *element_node* is used to store an element inside a structure. The information inside an element node includes function_ID which represents a graphics primitive function or an attribute setting function, the entry of this function parameter, and the number of bytes used to store the parameter. It is not difficult to find that this global linked list lets us access and modify each object in an easy way. For better understanding, Figure 18 gives the data structures for *structure node* and *element_node*. The set of programs which implements storage structure is given in appendix II. A sample program using storage structure is given in appendix III. This program is designed to continuously rotate a triangle about the y-axis and display it.

```
struct LIST_NODE {
            struct LIST_NODE  *prev;
            struct LIST_ NODE  *next;
            int    structure_ID;
            int    element_total;
            struct  ELEMENT *elementPtr;
    };      /* data structure for structure_node  */

    struct  ELEMENT {
        int    function_ID;
        int    param_size;
        char    *paramPtr;
        struct  ELEMENT *prev;
        struct  ELEMENT *next;
    };       /*  data structure for element_node  */
```

Figure 18. Data structures for structure_node and element_node.

OKLAHOMA STATE UNIVERSITY

5. Summary.

Gnat 95 is a good Ada program development environment. It provides a set of good features for interface with other programming languages. Among these, the representation specifications are those most important features deserved to be mentioned here; they are classified as length specification, enumeration type representation specification, record type representation, and address specification. We employ the above four types of representation specification and two types of interface pragmas to map the format and representation of objects in Ada to objects in C.

In this thesis, we have designed a free Ada 95 graphics package GNA95GP and implemented two major parts of it. GNA95GP is modeled after SRGP and PHIGS which are two standard graphics packages. So, it has most of the standard 2D graphics capabilities. The graphics packages in GNA95GP are organized in a way that a user familiar with Ada can understand and use it without much difficulty.

A PHIGS style storage structure will be a good addition to the present system. This is left as future work.

6. Bibliography

1. Booch, G., *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, ©1987.
2. James, D., Andries, V., and Steven, K., *Introduction to Computer Graphics*, Addison-Wesley Publishing Company, Inc., February, ©1994.
3. *Ada Quality and Style: Guidelines for Professional Programmers*, Software Productivity Consortium, Inc., Herndon, Virginia, December, ©1992.
4. *Ada 9X Reference Manual*, Intermetrics, Inc., Cambridge, Mass, ©1994.
5. Gart, M., *Interfacing Ada To C -- Solutions to Four Problems*, TRI-Ada'95 Conference Proceedings, pp. 28-34, November, ©1995.
6. Charles, P., *Programming Windows 3.1*, Microsoft Press, Redmond, Washington, ©1992.
7. *OpenGL(tm),* Web site: http://www.sgi.com/Products/Dev_environ_ds.html, maintained by Silicon Graphics company.
8. *List of published SC24 Standards: GKS-Graphical Kernel System*, Web site: http://www.cwi.nl/jTC1SC24/docs.html, maintained by the research institute CWI.
9. *Ada Bindings*, Web site: http://sw-eng.falls-church.va.us/AdaIC/source-code/Welcome.html, maintained by IIT Research Institute.
10. *Programmer's Hierarchical Interactive Graphics System (PHIGS),* Web site http://sw-eng.falls-church.va.us/AdaIC/source-code/bindings/binding95/html/section2.html, maintained by IIT Research Institute.

11 *Ada Bindings*, Web site: http://sw-eng.falls-church.va.us/AdaIC/source-code/bindings/binding95/binding95.txt, maintained by IIT Research Institute.

12. *DJGPP Frequently Asked Questions*, Web site: http://www.delorie.com/djgpp/faq/, maintained by Delorie Software.

13. GNAT: Free Ada 95 Compiler, Web site: http://wuarchive.wustl.edu/languages/ada/, maintained by Washington University.

14. *RSX Windows Development Kit*, ftp site: ftp.uni-bielefeld.de/pub/systems/msdos/misc, maintained by Delorie Software.

15. Enderle, G., Kansy, K., Pffaff, G., *Computer Graphics Programming: GKS-The Graphics Standard.* Springer-Verlag Heidelberg, New York, ©1987.

16. Hopgood, F. R. A., Duce, D. A., Gallop, J. R., Sutcliffe, D. C., *Introduction to the Graphical Kernel System,* Academic Press, Inc., Orlando, Florida, ©1986.

17. Scott, J, E., *Introduction to Interactive Computer Graphics,* John Wiley & Sons, Inc., ©1982.

18. Feldman, M. B., Elliot, K. B., *Ada 95: Problem solving and program design,* Addison-Wesley Publishing Company, Inc., ©1996.

19. Kosko, L., *PHIGS Reference Manual,* O'Reilly & Associates, Inc., Sebastopol, CA, ©1989.

20. David, F. R., *Computer Graphics Techniques: Theory and Practice,* Springer-Verlag New York Inc., ©1990.

21. James, F., Leisy, J., *Computer Graphics: The Principles Behind The Art And Science,* Franklin, Beedle & Associates, Irvine, CA, ©1989.

22. James, F., Andries, D., *Computer Graphics: Principle and Practice*, Addison-Wesley Publishing Company, Inc., ©1990.

23. Jagannathan, Mukund., *Multiple Parallel GKS Workstations*, MS Thesis, Comupter Science Department, Oklahoma State University, 1981.

24. Lan, L., Shang, K., George, K. M., *A 2D graphics package in Ada 95*, Proceedings of the 10th Annual ASEET Symposium, pp. 115-133, 1996.

25. Lewis, S. *Ada Implementation Of An X Window System Sever*, TRI-Ada' 89 Conference Proceedings, pp. 30-38, October, 1989.

26. Kuang, S. *Event handling in a GNAT 95 graphics package*, MS thesis in progress, Computer Science Department, Oklahoma State University.

Appendix_I

```
---------------------------------------------------------------------------------------------
-- Example application which uses GNA95GP to open a structure, set drawing
-- attributes and draw graphics primitive in screen.
---------------------------------------------------------------------------------------------
with init;
with objtype;
with Output;
with attribute;
with attrtype;
with structure;
procedure userMain is

use Output;
use objtype;
use init;
use structure;

p1, p2:       tagPoint;
X_list:       VErtexXCoordList(0..10);
Y_list:       VErtexYCoordList(0..10);
vertics:      VErtexList(0..10);
rect:         tagRectangle;
start_angle, end_angle: angleType;
i:            integer;
text:         String :="Ocean Picture";
--line_style: LineStyle;
pragma suppress(Range_Check,  On =>XCoord);
begin
  Gnat95gp_init;
  if Gna95gp_openStructure(1)=1 then
    attribute.gna95gp_setBackgroundColor(1);
    attribute.gna95gp_setColor(7);

    --draw the ocean
    p1 := makePoint(0, 479);
    p2 := makePoint(630, 180);
    attribute.gna95gp_setFillBitmapPattern(6);
    Gna95gp_FilledRectanglePoint(p1,p2);

    --draw the littile mounts.
    X_list(0) := 40;  X_list(1) := 60;
    X_list(2) := 80;  X_list(3) := 100;
```

```
X_list(4) := 145; X_list(5) := 160;
X_list(6) := 225; X_list(7) := 40;

Y_list(0) := 180; Y_list(1) := 135;
Y_list(2) := 155; Y_list(3) := 120;
Y_list(4) := 144; Y_list(5) := 128;
Y_list(6) := 180; Y_list(7) := 180;
for i in 0 .. 7 loop
 vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setColor(0);
Gna95gp_Polygon(8,  vertics);

--draw sun
attribute.gna95gp_setBackgroundColor(2); -- red
attribute.gna95gp_setColor(2);
rect := makeRect_Coord(120, 80, 160, 40);
p1 := makePoint(160, 60);
p2 := makePoint(160, 60);
Gna95gp_FilledEllipse(rect, p1, p2);

-- draw sun shing
--draw fat fish 1
attribute.gna95gp_setColor(5);
 rect := makeRect_Coord(300, 250, 350, 225),
p1 := makePoint(350, 225);
p2 := makePoint(350, 250);
Gna95gp_FilledEllipse(rect, p1, p2);
--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(5);
Gna95gp_DrawPixel(306, 233);

--draw  slim fish
X_list(0) := 0+100;  X_list(1) := 12+100,
X_list(2) := 22+100; X_list(3) := 24+100;
X_list(4) := 12+100; X_list(5) := 0+100;

Y_list(0) := 16+360; Y_list(1) := 11+360;
Y_list(2) := 22+360; Y_list(3) := 15+360;
Y_list(4) := 22+360; Y_list(5) := 16+360;
for i in 0 .. 5 loop
 vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setFillStyle(attrtype.SOLID);
```

```
attribute.gna95gp_setColor(6);  --purple
attribute.gna95gp_setLineWidth(1);
Gna95gp_Polygon(6,  vertics);

--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(4);
Gna95gp_DrawPixel(6+100, 16+360);
X_list(0) := 0+280;  X_list(1) := 12+280;
X_list(2) := 22+280; X_list(3) := 24+280;
X_list(4) := 12+280; X_list(5) := 0+280;

Y_list(0) := 16+260; Y_list(1) := 11+260;
Y_list(2) := 22+260; Y_list(3) := 15+260;
Y_list(4) := 22+260; Y_list(5) := 16+260;
for i in 0 .. 5 loop
  vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setFillStyle(attrtype.SOLID);
attribute.gna95gp_setColor(6);
attribute.gna95gp_setLineWidth(1);
Gna95gp_Polygon(6,  vertics);
--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(4);
Gna95gp_DrawPixel(6+280, 16+260);

X_list(0) := 0+520;  X_list(1) := 12+520;
X_list(2) := 22+520; X_list(3) := 24+520;
X_list(4) := 12+520; X_list(5) := 0+520;

Y_list(0) := 16+390; Y_list(1) := 11+390;
Y_list(2) := 22+390; Y_list(3) := 15+390;
Y_list(4) := 22+390; Y_list(5) := 16+390;
for i in 0 .. 5 loop
  vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setFillStyle(attrtype.SOLID);
attribute.gna95gp_setColor(6);
attribute.gna95gp_setLineWidth(1);
Gna95gp_Polygon(6,  vertics);

--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(4);
```

```
Gna95gp_DrawPixel(6+520, 16+390);

X_list(0) := 0+420;  X_list(1) := 12+420;
X_list(2) := 22+420; X_list(3) := 24+420;
X_list(4) := 12+420; X_list(5) := 0+420;
Y_list(0) := 16+310; Y_list(1) := 11+310;
Y_list(2) := 22+310; Y_list(3) := 15+310;
Y_list(4) := 22+310; Y_list(5) := 16+310;
for i in 0 .. 5 loop
  vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setFillStyle(attrtype.SOLID);
attribute.gna95gp_setColor(6);
attribute.gna95gp_setLineWidth(1);
Gna95gp_Polygon(6,  vertics);

--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(4);
Gna95gp_DrawPixel(6+420, 16+310);

--draw big fish
X_list(0) := 0+360;  X_list(1) := 32+360;
X_list(2) := 27+360; X_list(3) := 34+360;
X_list(4) := 34+360; X_list(5) := 27+360;
X_list(6) := 32+360; X_list(7) := 0+360;
Y_list(0) := 16+335; Y_list(1) := 0+335;
Y_list(2) := 14+335; Y_list(3) := 9+335;
Y _list(4) := 24+335; Y_list(5) := 19+335;
Y_list(6) := 32+335; Y_list(7) := 16+335;
for i in 0 .. 7 loop
  vertics(i) := makePoint(X_list(i), Y_list(i));
end loop;
attribute.gna95gp_setLineWidth(1);
attribute.gna95gp_setColor(5);
Gna95gp_Polygon(8,  vertics);
--draw eye
attribute.gna95gp_setColor(0);
attribute.gna95gp_setLineWidth(4);
Gna95gp_DrawPixel(4+360, 17+335);

--draw boat
X_list(0) := 500; X_list(1) := 550;
X_list(2) := 540; X_list(3) := 510;
X_list(4) := 500;
```

```
    Y_list(0) := 155; Y_list(1) := 155;
    Y_list(2) := 180; Y_list(3) := 180;
    Y_list(4) := 155;
    for i in 0 .. 4 loop
      vertics(i) := makePoint(X_list(i), Y_list(i));
    end loop;
    attribute.gna95gp_setLineWidth(1);
    attribute.gna95gp_setColor(0);
    Gna95gp_Polygon(5,  vertics);

    X_list(0) := 510; X_list(1) := 515;
    X_list(2) := 515, X_list(3) := 510;
    X_list(4) := 510;
    Y_list(0) := 143; Y_list(1) := 143;
    Y_list(2) := 155; Y_list(3) := 155;
    Y_list(4) := 143;
    for i in 0 .. 4 loop
      vertics(i) := makePoint(X_list(i), Y_list(i));
    end loop;
    attribute.gna95gp_setColor(0);
    Gna95gp_Polygon(5,  vertics);
    --output text "OCEAN picture"
    attribute.gna95gp_setBackgroundColor(1);
    attribute.gna95gp_setColor(4);
    p1 := makePoint(400, 110);
   Gna95gp_text(p1, text);
   Gna95gp_closeStructure;
end if
end userMain;
pragma Export(C, userMain, "UserMain");
```

Appendix_II

```
-------------------------------------------------------------------------------------------------------
-- file name: structure.ads
-- Ada package for structure storage in GNA95GP. Functions to manage a structure
-- are included in this package.
-------------------------------------------------------------------------------------------------------
package structure  is
  -- open an structure.
  function Gna95gp_openStructure(structure_id  integer) return integer;
  -- close an structure.
  procedure Gna95gp_closeStructure;
  -- delete an structure.
  function Gna95gp_deleteStructure(structure_id: integer) return integer;
  -- delete an element.
  procedure Gna95gp_deleteElement;
  -- set current index.
  procedure Gna95gp_setElementIndex(index: integer);
  -- move current index backward or forward.
  procedure Gna95gp_OffsetElementIndex(offset: integer);
  -- for temporary use only
  procedure Gna95gp_waitkeyEvent;
end structure;

-------------------------------------------------------------------------------------------------------
-- file name structure.adb
-- Implementation for structure.ads
-------------------------------------------------------------------------------------------------------
package body structure  is
  function WIN_openStructure(structure_id: integer) return integer;
  pragma import (C,WIN_openStructure, "Win_OpenStructure");
  function Gna95gp_openStructure(structure_id: integer) return integer is
  result: integer;
  begin
    result :=WIN_openStructure(structure_id);
    return result;
  end Gna95gp_openStructure;


  procedure WIN_closeStructure;
  pragma import (C,WIN_closeStructure, "Win_CloseStructure");
  procedure Gna95gp_closeStructure is
  begin
    WIN_closeStructure;
```

```
 end Gna95gp_closeStructure;

 function WIN_deleteStructure(structure_id: integer) return integer;
 pragma import (C,WIN_deleteStructure, "Win_deleteStructure");
 function Gna95gp_deleteStructure(structure_id: integer) return integer is
 begin
  return(WIN_deleteStructure(structure_id));
 end Gna95gp_deleteStructure;

 procedure WIN_deleteElement;
 pragma import (C,WIN_deleteElement, "Win_deleteElement");
 procedure Gna95gp_deleteElement is
 begin
  WIN_deleteElement;
 end Gna95gp_deleteElement;

 procedure WIN_setElementIndex(index: integer);
 pragma import (C,WIN_setElementIndex, "Win_setElementIndex");
 procedure Gna95gp_setElementIndex(index: integer) is
 begin
  WIN_setElementIndex(index);
 end Gna95gp_setElementIndex;

 procedure WIN_OffsetElementIndex(offset: integer);
 pragma import (C,WIN_OffsetElementIndex, "Win_OffsetElementIndex");
 procedure Gna95gp_OffsetElementIndex(offset: integer) is
 begin
  WIN_OffsetElementIndex(offset);
 end Gna95gp_OffsetElementIndex;

 -- temporary use only
 procedure WIN_waitkeyEvent;
 pragma import (C,WIN_waitkeyEvent, "Win_waitkeyEvent");
 procedure Gna95gp_waitkeyEvent is
 begin
  WIN_waitkeyEvent;
 end Gna95gp_waitkeyEvent;
end structure;

/*******************************************************************
file name:  storage.h
Head file for the underlying C program used to implement storage structure. It defines
constants, macros, and data structures  for structure storage in GNA95GP.
*******************************************************************/
/*  define function's ID  */
```

```
#define DrawPixel_ID                1
#define DrawMarker_ID               2
#define lineCoord_ID                3
#define line_ID                     4
#define RectanglePoint_ID           5
#define Rectangle_ID                6
#define polyLineCoord_ID            7
#define PolyLine_ID                 8
#define Polygon_ID                  9
#define FilledRectanglePoint_ID     10
#define FilledRectangle_ID          11
#define ellipseArc_ID               12
#define FilledEllipse_ID            13
#define text_ID                     14

#define setLineWidth_ID             50
#define setLineStyle_ID             51
#define setColor_ID                 52
#define setBackgroundColor_ID       53
#define setFillStyle_ID             54
#define setFillBitmapPattern_ID     55
#define setWriteMode_ID             56

#define FILLED                      0
#define NON_FILLED                  1
#define True                        1
#define False                       0

/********* check validation of dynamic allocated memory **/
#define CheckValid(ptr)                                   \
  if ((ptr)==NULL)  {                                     \
   MessageBox(NULL, "failure of memeory allocation", "check", MB_OK); \
   InvalidateRect(hwndMain, NULL, TRUE);                  \
  }

/*  data structure for ELEMENT node */
struct  ELEMENT {
     int    function_ID;           /* function ID */
     int    param_size;            /*  size of parameter for the function */
     char    *paramPtr;            /*  pointer to the parameter of the function */
     struct  ELEMENT *prev;        /*  pointer to the immediately precceded
                                       ELEMENT  node */
     struct  ELEMENT *next;        /*  pointer to the immediately followed
                                      ELEMENT node */
 };
```

```
struct LIST {
      struct LIST *prev;               /* pointer to the immediately precceded
                                          LIST node */
      struct LIST *next;               /* pointer to the immediately followed
                                          LIST node */
      int    structure_ID;             /* structure ID */
      int    element_total;            /* total number of elements inside the
                                          structure */
      struct ELEMENT *elementPtr; /* pointer to the first ELEMENT node */
 };

/* data structure for parameter of the drawing line function */
 struct param_lineCoord {
   int x1;
   int y1;
   int x2;
   int y2;
  };

/* data structure for parameter of the drawing pixel function */
 struct param_DrawPixel {
   int x;
   int y;
 };

/* data structure for parameter of the drawing ellipse function */
 struct param_ellipse {
   int left;
   int top;
   int right;
   int bottom;
   int x1,
   int y1;
   int x2;
   int y2;
 };

/* data structure for parameter of the drawing marker function */
 struct param_DrawMarker {
  int x;
  int y;
  int markerSize;
  int markerStyle;
 };
```

```
/* data structure for parameter of the drawing polygon function */
struct param_poly{
   int vertics;
   int *ptr;      /* point to the X, Y, coordinate array */
 };
/* data structure for parameter of the output text string function */
struct param_text {
   int x;
   int y;
   int stringSize;
   char *ptr;   /*  point to the text string */
 };

/* data structure for parameter of the setting  line width function */
struct param_setLineWidth {
  int lineWidth;
 };

/* data structure for parameter of the setting  line style function */
struct param_setLineStyle{
  int lineStyle;
 };

/* data structure for parameter of the setting  front color function */
struct param_setColor{
  int color;
 };

/* data structure for parameter of the setting  filling style function */
struct param_setFillStyle {
 int fillStyle;
 };

/* data structure for parameter of the setting  filling bitmap pattern function */
 struct param_setFillBitmapPattern {
 int patternIndex;
 };

/* data structure for parameter of the setting write mode function */
 struct  param_setWriteMode {
  int mode;
 };

/*****************************************************************
```

```
file name:  lnklist.c.
The C part of implementation of structure storage. It uses a global Linked list to
 store all the existing STRUCTUREs and their associated ELEMENTs.
*****************************************************************/
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "storage.h"

#define NORMAL   1
#define ABNORMAL 0

extern  HDC hdc;
extern HWND hwndMain;

struct LIST *List_entry;               /* entry to the global linked list */
struct LIST *Current_structure;        /*  pointer to the current opened structure. */
int    Current_index;                  /*  current index. */
int    OpenFlag;
int    Structure_ID;


int Win_OpenStructure(int ID);
int Win_deleteStructure(int ID);
void Win_CloseStructure();
void Win_deleteElement();
void InsertTo(struct ELEMENT *newNode);
void Storage_init();
void end_proc();
void Win_setElementIndex(int index);
void Win_OffsetElementIndex(int offset);

void ClearScreen();
void Display_AllStructure();

void Call_DrawPixel(struct ELEMENT *ptr);
void  Call_DrawMarker(struct ELEMENT *elementPtr);
void Call_lineCoord(struct ELEMENT *ptr);
void  Call_line(struct ELEMENT *elementPtr);
void Call_RectanglePoint(struct ELEMENT *elementPtr);
void  Call_Rectangle(struct ELEMENT *elementPtr);
void  Call_FilledRectanglePoint(struct ELEMENT *elementPtr);
void  Call_FilledRectangle(struct ELEMENT *elementPtr);
void  Call_ellipseArc(struct ELEMENT *elementPtr);
void  Call_FilledEllipse(struct ELEMENT *elementPtr);
```

```
void Call_polyLineCoord(struct ELEMENT *ptr);
void  Call_PolyLine(struct ELEMENT *elementPtr);
void  Call_Polygon(struct ELEMENT *elementPtr);
void  Call_text(struct ELEMENT *elementPtr);


void  Call_setLineWidth(struct ELEMENT *elementPtr);
void  Call_setLineStyle(struct ELEMENT *elementPtr);
void  Call_setColor(struct ELEMENT *elementPtr);
void  Call_setBackgroundColor(struct ELEMENT *elementPtr),
void  Call_setFillStyle(struct ELEMENT *elementPtr);
void  Call_setFillBitmapPattern(struct ELEMENT *elementPtr);
void  Call_setWriteMode(struct ELEMENT *elementPtr);

void  set_DefaultAttribute();

/*****************************************************************
Function:      Initialization of the global variables.
Parameters:    None.
*****************************************************************/
void Storage_init()
{
 List_entry = NULL;
 Current_structure = NULL;
 Current_index=0;
 OpenFlag = False;
}

/*****************************************************************
Function:      To open an structure.
Parameters:    ID -- structure ID.
*****************************************************************/
int Win_OpenStructure(int ID)
{
 struct LIST *ptr, *newNode;
 int Find;
 int i;

 if (OpenFlag == True) return(ABNORMAL);
 OpenFlag = True;
 /* begin search for the structure in this link_list */
 ptr = List_entry;
 Find = False;
 if (ptr!=NULL)
  for (i=0; ; i++) {
```

```
    if (ptr->structure_ID == ID) {
        Find  = True;
        break;
    }
    if (ptr->next==NULL) break;   /* break when reach the last structure */
    ptr = ptr->next;
   }
 if (Find == False )  { /*  new structure need to be created */
  newNode = malloc(sizeof(struct LIST));
  CheckValid(newNode);
  newNode->structure_ID = ID;
  newNode->element_total = 0;
  newNode->elementPtr = NULL;
  Current_structure = newNode;
  Current_index = 0;
  if (ptr==NULL) {  /* at first, link list is empty */
    newNode->prev = NULL;
    newNode->next = NULL;
    List_entry = newNode;
  }
  else { /* link list is not empty */
    ptr->next = newNode;
    newNode->prev = ptr;
    newNode->next = NULL;
  }
 }
 else { /* find ==true; the structure already existed */
   Current_structure = ptr;
   Current_index = ptr->element_total;
 }
 Structure_ID = ID;
 return(NORMAL);
}

/*****************************************************************
Function:     Insert an ELEMENT node to the current structure.
Parameters:   newNode -- pointer to the ELEMENT node to be inserted
*****************************************************************/
void InsertTo(struct ELEMENT *newNode)
{
 struct ELEMENT *elementPtr;
 int i;
 if (OpenFlag==True) {

  elementPtr = Current_structure->elementPtr;
```

```
  for (i=1; i<Current_index; i++)
   elementPtr = elementPtr->next;
  if (Current_index == Current_structure->element_total) {
  /* insert to the end of this structure */
    if (Current_index ==0) {  /* no element inside this structucture */
     Current_structure->elementPtr = newNode;
     newNode->prev = NULL;
     newNode->next = NULL;
    }
    else {
     elementPtr->next = newNode;
     newNode->prev = elementPtr;
     newNode->next = NULL;
    }
  }
  else {
    if (Current_index == 0) { /*insert as the first element inside this structure*/
     newNode->next = Current_structure->elementPtr;
     Current_structure->elementPtr->prev = newNode;
     Current_structure->elementPtr = newNode;
     newNode->prev = NULL,
    }
    else {
     newNode->next = elementPtr->next;
     elementPtr->next->prev = newNode;
     elementPtr->next = newNode;
     newNode->prev = elementPtr;
    }
  }
  Current_index++;                        /* increase the current index by 1. */
  Current_structure->element_total++;     /*  increase the total number of the
                                          ELEMENT  in current structure  */
 }
}

/****************************************************************
Function:      Free the space allocated to an ELEMENT node
Parameters:    elementPtr -- Pointer to the ELEMENT node.
****************************************************************/
void freeElementSpace(struct ELEMENT *elementPtr)
{
    struct param_poly *parameter;
    struct param_text *t_parameter;

    switch (elementPtr->function_ID) {
```

```
      case DrawPixel_ID:
      case DrawMarker_ID:
      case lineCoord_ID:
      case line_ID:
      case RectanglePoint_ID:
      case Rectangle_ID:
      case FilledRectanglePoint_ID:
      case FilledRectangle_ID:
      case ellipseArc_ID:
      case FilledEllipse_ID:
        break;
      case polyLineCoord_ID:
      case PolyLine_ID:
      case Polygon_ID:
        parameter = (struct param_poly *)elementPtr->paramPtr;
        free(parameter->ptr);
        break;
      case text_ID:
        t_parameter = (struct param_text *)elementPtr->paramPtr;
        free(t_parameter->ptr);
        break;
      default:
       break;
    }
    free(elementPtr->paramPtr);
    free(elementPtr);
}

/******************************************************************
Function:      Delete all the elements inside a structure.
Parameters:    FirstElement -- Pointer to the first ELEMENT node.
               total_element -- total number of elements inside a structure
******************************************************************/
void DeleteAllElements(struct ELEMENT *FirstElement, int total_element)
{
  struct ELEMENT *elementPtr, *nextElementPtr;
  int j;

  elementPtr = FirstElement;
  /*interation for every element in this structure */
  for (j=0; j<total_element; j++) {
   nextElementPtr = elementPtr->next;
   freeElementSpace(elementPtr);
   elementPtr = nextElementPtr;
  }
```

```
}

/*****************************************************************
Function:       To close an opened structure.
Parameters:     None.
*****************************************************************/
void Win_CloseStructure()
{
 if (OpenFlag==True) {
   OpenFlag = False;
   ClearScreen();
   Display_AllStructure();
  }
}

/*****************************************************************
Function:       Clear the client area of the screen.
Parameters:     None.
*****************************************************************/
void ClearScreen()
{
 RECT rect;
 HPEN  hpenOld;
 HBRUSH hbrOld;

 GetClientRect(hwndMain, &rect);

 hpenOld = SelectObject(hdc, GetStockObject(NULL_PEN)),
 hbrOld = SelectObject(hdc, GetStockObject(WHITE_BRUSH));
 Rectangle(hdc, rect.left, rect.top, rect.right, rect.bottom);
 SelectObject(hdc,hpenOld);
 SelectObject(hdc,hbrOld);
}

/*****************************************************************
Function:       Display all the existing structures created by the user application.
Parameters:     None.
*****************************************************************/
void Display_AllStructure()
{
 struct LIST *nodePtr;
 struct ELEMENT *elementPtr;
 int i,j;
 char   chBuf[80];
```

```
set_DefaultAttribute(),
nodePtr = List_entry;
/* interation for every strcuture in the link list */
for (i=0; ; i++) {
 if (nodePtr==NULL) break;
 elementPtr = nodePtr->elementPtr;
 /*interation for every element in this structure */
 for (j=0; j < nodePtr->element_total; j++) {
   switch (elementPtr->function_ID) {
     case DrawPixel_ID:
         Call_DrawPixel(elementPtr);
         break;
     case DrawMarker_ID:
         Call_DrawMarker(elementPtr);
         break;
     case lineCoord_ID:
         Call_lineCoord(elementPtr);
         break;
     case line_ID:
         Call_line(elementPtr);
         break;
     case RectanglePoint_ID:
         Call_RectanglePoint(elementPtr);
         break;
     case Rectangle_ID:
         Call_Rectangle(elementPtr);
         break;
     case FilledRectanglePoint_ID:
         Call_FilledRectanglePoint(elementPtr);
         break;
     case FilledRectangle_ID:
         Call_FilledRectangle(elementPtr);
         break;
     case ellipseArc_ID:
         Call_ellipseArc(elementPtr);
         break;
     case FilledEllipse_ID:
         Call_FilledEllipse(elementPtr);
         break;
     case polyLineCoord_ID:
         Call_polyLineCoord(elementPtr);
         break;
     case PolyLine_ID.
         Call_PolyLine(elementPtr);
         break;
```

```
      case Polygon_ID:
        Call_Polygon(elementPtr);
        break;
      case text_ID:
        Call_text(elementPtr);
        break;
      case setLineWidth_ID:
        Call_setLineWidth(elementPtr);
        break;
      case setLineStyle_ID:
        Call_setLineStyle(elementPtr);
        break;
      case setColor_ID:
        Call_setColor(elementPtr);
        break;
      case setBackgroundColor_ID:
        Call_setBackgroundColor(elementPtr);
        break;
      case setFillStyle_ID:
        Call_setFillStyle(elementPtr);
        break;
      case setFillBitmapPattern_ID:
        Call_setFillBitmapPattern(elementPtr);
        break,
      case setWriteMode_ID:
        Call_setWriteMode(elementPtr);
        break;
      default.
        break;
    }
    elementPtr = elementPtr->next,
   } /* end of j loop */
   nodePtr = nodePtr->next;
   set_DefaultAttribute();
  } /* end of i loop  */
}

/*****************************************************************
Function:     Termination handling, free all the space allocated to the linked list.
Parameters:   None.
*****************************************************************/
void end_proc()
{
 struct ELEMENT *elementPtr;
 struct LIST *nodePtr, *nextNodePtr;
```

```
 int i;

 nodePtr = List_entry;
 for (i=0; ; i++) { /* interation for every structure in the link list*/
  if (nodePtr == NULL) break; /* reach the end of the structure link list */
  elementPtr = nodePtr->elementPtr;
  DeleteAllElements(elementPtr, nodePtr->element_total);
  nextNodePtr = nodePtr->next;
  free(nodePtr);
  nodePtr = nextNodePtr;
 } /* end of i loop */
}

/*****************************************************************
Function:       To delete a structure given by the ID.
Parameters:     ID -- structure ID.
*****************************************************************/
int Win_deleteStructure(int ID)
{
 struct LIST *nodePtr;
 int Find;
 int i;

 if (OpenFlag == True) return(ABNORMAL); /* It is true that no opened structure */
 /* begin search for the structure in this link_list */
 nodePtr = List_entry;
 Find = False;
 for (i=0; ; i++) {
     if (nodePtr==NULL) break; /*break when reach the end of the structure*/
     if (nodePtr->structure_ID == ID) {
       Find = True;
       break;
     }
     nodePtr = nodePtr->next;
 }
 if (Find == False) return(ABNORMAL);
 else { /* found */
  DeleteAllElements(nodePtr->elementPtr, nodePtr->element_total);
  if (nodePtr->prev==NULL) { /* head of the link list */
     List_entry = nodePtr->next;
     if (nodePtr->next!=NULL)
        nodePtr->next->prev = NULL;
  }
  else {
     nodePtr->prev->next = nodePtr->next;
```

```
      if (nodePtr->next!=NULL)
        nodePtr->next->prev = nodePtr->prev;
    }
    free(nodePtr);
    ClearScreen();
    Display_AllStructure();
    return(NORMAL);
  }
}

/*****************************************************************
Function:      Delete the current element inside the opened structure.
Parameters:    None.
*****************************************************************/
void Win_deleteElement()
{
  struct ELEMENT *elementPtr;
  int i;
  if (OpenFlag==True) {
    elementPtr = Current_structure->elementPtr;
    for (i=1; i<Current_index; i++)
      elementPtr = elementPtr->next;
    if (Current_index!=0) {
        if (Current_index == Current_structure->element_total) {
        /* delete the last element inside the structure */
          if (Current_index==1) /* only one element inside the structure*/
            Current_structure->elementPtr = NULL;
          else
            elementPtr->prev->next = NULL;
        }
        else {
          if (Current_index==1) { /*delete the first element inside the structure*/
            Current_structure->elementPtr = elementPtr->next;
            elementPtr->next->prev = NULL;
          }
          else {
            elementPtr->prev->next = elementPtr->next;
            elementPtr->next->prev = elementPtr->prev;
          }
        }
        freeElementSpace(elementPtr);
        Current_structure->element_total--;
        Current_index--;
    }
  }
```

```
}

/*****************************************************************
Function:      Set the current index which used to identify the current element.
Parameters:    index -- index used to identify the current element.
*****************************************************************/
void Win_setElementIndex(int index)
{

 if (OpenFlag==True)
  if (index < 0)
    Current_index = 0;
  else if (index > Current_structure->element_total)
        Current_index = Current_structure->element_total;
      else
        Current_index = index;
}

/*****************************************************************
Function:      used to move current index forward or backward.
Parameters:    offset -- move steps from the current element.
*****************************************************************/
void Win_OffsetEle
+mentIndex(int offset)
{
 if (OpenFlag==True)
  if ((Current_index+offset) < 0)
    Current_index = 0;
  else if ((Current_index+offset) > Current_structure->element_total)
        Current_index = Current_structure->element_total;
      else
        Current_index = Current_index + offset;
}

/*****************************************************************
file name: outpc.c
It contains the drawing primitive functions and attribute setting functions for GNA95GP.
*****************************************************************/
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "gna_gp.h"
#include "storage.h"
```

```
extern HWND hwndMain;
extern  HDC hdc;
extern  RGB_Color Cur_Color;

/******* export functions to Ada package************/
void WinDrawPixel(HWND Whnd, int x, int y);
void WinLineCoord(HWND Whnd, int x1, int y1, int x2, int y2);
void WinPolyLineCoord(HWND Whnd, int vertexCount, int* xList,int* yList);
void WinPolygon(HWND Whnd, int vertexCount, int* xList,int* yList);
/*void WinEllipseArc(RECT rect, double startAngle, double endAngle); */


void WinEllipseArc(HWND Whnd,int left, int top, int right, int bottom,
                        int x1, int y1, int x2, int y2);

void Win_FilledEllipse(HWND Whnd,int left, int top, int right, int bottom,
                        int x1, int y1, int x2, int y2);
void Win_drawMarker(HWND Whnd, int x, int y, int markerSize, int markerStyle);
void Win_Text(HWND Whnd, int x, int y, char *text, int textSize);
void polyLineCoord(int vertexCount, int* xList,int* yList);
void InsertTo(struct ELEMENT *elementPtr);
extern struct LIST *List_entry;
extern struct LIST *Current_structure;
extern int    Current_index;

/*****************************************************************
Function:      Create a ELEMENT node for the pixel drawing function, insert this
                node in the opened structure.
Parameters:    Whnd -- window's handler
               (x,y) -- screen position where the pixel will be drawn.
*****************************************************************/
void WinDrawPixel(HWND Whnd, int x, int y)
{
 struct ELEMENT *element;
 struct param_DrawPixel *parameter;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
 CheckValid(element);
 element->function_ID = DrawPixel_ID;

 parameter =(struct param_DrawPixel *)malloc(sizeof(struct param_DrawPixel));
 CheckValid(parameter);
 parameter->x = x;
 parameter->y = y;
 element->paramPtr = (char *)(parameter);
```

```
 InsertTo(element);
}

/********************************************************************
Function:      Create a ELEMENT node for the line drawing function, insert this
                node in the opened structure.
Parameters:    Whnd -- window's handler.
               (x1,y1) -- start point for the line.
               (x2,y2) -- end point for the line.
********************************************************************/
void WinLineCoord(HWND Whnd, int x1, int y1, int x2, int y2)
{
 struct ELEMENT *element;
 struct param_lineCoord *parameter;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
 CheckValid(element);
 element->function_ID = lineCoord_ID;

 parameter =(struct param_lineCoord *)malloc(sizeof(struct param_lineCoord));
 CheckValid(parameter);
 parameter->x1 = x1;
 parameter->y1 = y1;
 parameter->x2 = x2;
 parameter->y2 = y2,
 element->paramPtr = (char *)(parameter);
 InsertTo(element);
}

/********************************************************************
Function:      Create a ELEMENT node containing the polygon drawing or
               polyline drawing function, insert this node in the opened structure.
Parameters:    filling_flag -- used to indicate if it is a polyline function or
               a polygon function.
               vertexCount -- number of vertex.
               xList -- a array containing x coordinates for this polygon or polyline.
               yList -- a array containing y coordinates for this polygon or polyline.
********************************************************************/
void polyShape(int filled_flag, int vertexCount, int* xList,int* yList)
{
 struct ELEMENT *element;
 struct param_poly *parameter;
 int i;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
```

```
 CheckValid(element);
 if (filled_flag==FILLED)
   element->function_ID = Polygon_ID;
 else
   element->function_ID = polyLineCoord_ID;

 parameter = (struct param_poly *)malloc(sizeof(struct param_poly));
 CheckValid(parameter);
 parameter->vertics = vertexCount;
 parameter->ptr =(int *)calloc(2*vertexCount, sizeof(int));
 CheckValid(parameter->ptr);
 for (i=0; i<vertexCount; i++)
     parameter->ptr[i] = xList[i];
 for (i=0; i<vertexCount; i++)
     parameter->ptr[vertexCount+i] = yList[i];
 element->paramPtr = (char *)(parameter);
 InsertTo(element);
}

/* draw polyline */
void
WinPolyLineCoord(HWND Whnd, int vertexCount, int* xList,int* yList)
{
  polyShape(NON_FILLED, vertexCount, xList, yList);
}


/*****************************************************************
Function:       Drawing a polyline
Parameters:     vertexCount -- number of vertex.
                xList -- a array containing x coordinates for this polyline.
                yList -- a array containing y coordinates for this polyline.

*****************************************************************/
void polyLineCoord(int vertexCount, int* xList,int* yList)
{
  POINT vertexList[100];
  int i;

  for (i=0; i<vertexCount; i++) {
   vertexList[i].x = xList[i];
   vertexList[i].y = yList[i];
  }
  Polyline(hdc, vertexList, vertexCount);
}
```

```
/*****************************************************************
Function:     Drawing a polygon
Parameters:   vertexCount -- number of vertex.
              xList -- a array containing x coordinates for this polygon.
              yList -- a array containing y coordinates for this polygon.

*****************************************************************/
void polygon(int vertexCount, int* xList,int* yList)
{
  POINT vertexList[100];
  int i;

  for (i=0; i<vertexCount; i++) {
    vertexList[i].x = xList[i];
    vertexList[i].y = yList[i];
  }
  Polygon(hdc, vertexList, vertexCount);
}

void WinPolygon(HWND Whnd, int vertexCount, int* xList,int* yList)
{
  polyShape(FILLED, vertexCount, xList, yList);
}

/*****************************************************************
Function:     Create a ELEMENT node containing the ellipse drawing or
              ellipse arc drawing function, insert this node in the opened structure.
Parameters:   filling_flag -- used to indicate if it is an ellipse drwaing function or
              an ellipse arc drawing function.
              (left, top, right, bottom)-- to define the external rectangle of this ellipse
              (x1,y1) -- a point in the start edge of the ellipse.
              (x2,y2) -- a point in the end edge of the ellipse.
*****************************************************************/
void ellipse(int filled_flag,int left, int top, int right, int bottom,
                    int x1, int y1, int x2, int y2)
{
 struct ELEMENT *element;
 struct param_ellipse *parameter;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
 CheckValid(element);
 if (filled_flag == NON_FILLED)
   element->function_ID = ellipseArc_ID;
 else
```

```
  element->function_ID = FilledEllipse_ID;
 parameter =(struct param_ellipse *)malloc(sizeof(struct param_ellipse));
 CheckValid(parameter);
 parameter->left = left;
 parameter->top = top;
 parameter->right = right;
 parameter->bottom = bottom;
 parameter->x1 = x1;
 parameter->y1 = y1;
 parameter->x2 = x2;
 parameter->y2 = y2;
 element->paramPtr = (char *)(parameter);
 InsertTo(element);
}

/****************************************************************
Function:     Drawing an ellipse arc
Parameters:   Whnd -- windows handler.
              (left, top, right, bottom)-- to define the external rectangle of this ellipse arc.
              (x1,y1) -- a point in the start edge of the ellipse arc.
              (x2,y2) -- a point in the end edge of the ellipse arc.
****************************************************************/
void WinEllipseArc(HWND Whnd,int left, int top, int right, int bottom,
                        int x1, int y1, int x2, int y2)
{
 ellipse(NON_FILLED, left, top, right, bottom, x1, y1, x2, y2);
}

/****************************************************************
Function:     Drawing an ellipse.
Parameters:   Whnd -- windows handler.
              (left, top, right, bottom)-- to define the external rectangle of this ellipse.
              (x1,y1) -- a point in the start edge of the ellipse.
              (x2,y2) -- a point in the end edge of the ellipse.
****************************************************************/
void Win_FilledEllipse(HWND Whnd,int left, int top, int right, int bottom,
                            int x1, int y1, int x2, int y2)
{
 ellipse(FILLED, left, top, right, bottom, x1, y1, x2, y2);
}

/****************************************************************
Function:     Drawing a marker.
Parameters:   (x,y) --  screen position for the marker.
              markerSize -- size for this marker.
```

```
              markerStyle -- marker style.
*******************************************************************/
void drawMarker(int x, int y, int markerSize, int markerStyle)
{
 int xlist[6];
 int ylist[6];

 if (markerSize==1) {
  MoveTo (hdc, x, y);
  LineTo (hdc,x+1, y);
 }
 else if (markerStyle == MARKER_CIRCLE) {
      if ((markerSize%2) ==0)   /* even */
        Pie(hdc, x-(int)(markerSize/2),
          y-(int)(markerSize/2), x+(int)(markerSize/2)-1,
          y+(int)(markerSize/2)-1, x+(int)(markerSize/2)-1,
          y, x+(int)(markerSize/2)-1, y);
      else
        Pie(hdc, x-(int)((markerSize-1)/2),
          y-(int)((markerSize-1)/2), x+(int)((markerSize-1)/2),
          y+(int)((markerSize-1)/2), x+(int)((markerSize-1)/2),  y,
                    x+(int)((markerSize-1)/2),  y);

    }
    else {   /* markerStyle == MARKER_RECTANGLE */
      if ((markerSize%2) ==0) {
        xlist[0] = x-(int)(markerSize/2) +1;
        xlist[1] = x-(int)(markerSize/2)+1;
        xlist[2] = x+(int)(markerSize/2);
        xlist[3] = x+(int)(markerSize/2);
        xlist[4] = xlist[0];
        ylist[0] = y-(int)(markerSize/2)+1;
        ylist[1] = y+(int)(markerSize/2);
        ylist[2] = y+(int)(markerSize/2);
        ylist[3] = y-(int)(markerSize/2)+1;
        ylist[4] = ylist[0];
      }
      else {
        xlist[0] = x-(int)((markerSize-1)/2);
        xlist[1] = x-(int)((markerSize-1)/2);
        xlist[2] = x+(int)((markerSize-1)/2);
        xlist[3] = x+(int)((markerSize-1)/2);
        xlist[4] = xlist[0];
        ylist[0] = y-(int)((markerSize-1)/2);
        ylist[1] = y+(int)((markerSize-1)/2);
```

```
        ylist[2] = y+(int)((markerSize-1)/2);
        ylist[3] = y-(int)((markerSize-1)/2),
        ylist[4] = ylist[0];
      }
      polygon(5, xlist, ylist);
    }
}

/*****************************************************************
Function:     Create an ELEMENT node containing the marker drawing
              function, insert this node in the opened structure.
Parameters:   Whnd -- windows handler.
              (x,y) --  screen position for the marker.
              markerSize -- size for this marker.
              markerStyle -- marker style.
*****************************************************************/
void Win_drawMarker(HWND Whnd, int x, int y, int markerSize, int markerStyle)
{
 struct ELEMENT *element;
 struct param_DrawMarker *parameter;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
 CheckValid(element);
 element->function_ID = DrawMarker_ID;
 parameter =(struct param_DrawMarker *)malloc(sizeof(struct param_DrawMarker));
 CheckValid(parameter);
 parameter->x = x;
 parameter->y = y;
 parameter->markerSize = markerSize;
 parameter->markerStyle = markerStyle;
 element->paramPtr = (char *)(parameter);
 InsertTo(element);
}

/*****************************************************************
Function:     Create an ELEMENT node containing a text string output function,
              insert this node in the opened structure.
Parameters:   Whnd -- window's handler.
              (x,y) -- screen position for this text string.
               text -- pointer to the text string.
               textSize -- the length of the text string.
*****************************************************************/
void Win_Text(HWND Whnd, int x, int y, char *text, int textSize)
{
 struct ELEMENT *element;
```

```
 struct param_text *parameter;

 element = (struct ELEMENT *)malloc(sizeof(struct ELEMENT));
 CheckValid(element);
 element->function_ID = text_ID;
 parameter =(struct param_text *)malloc(sizeof(struct param_text));
 CheckValid(parameter);
 parameter->x = x;
 parameter->y = y;
 parameter->stringSize = textSize;
 parameter->ptr =(char *)calloc(textSize, sizeof(char));
 CheckValid(parameter->ptr);
 memcpy(parameter->ptr, text, textSize);
 element->paramPtr = (char *)(parameter);
 InsertTo(element);
}
/******************************************************************
Function:     Call the pixel drawing function whose ID is stored in the ELEMENT node.
Parameters:   ptr -- pointer to the ELEMENT node.
******************************************************************/

void Call_DrawPixel(struct ELEMENT *ptr)
{
 struct param_DrawPixel *parameter;

 parameter = (struct param_DrawPixel *)(ptr->paramPtr);
 MoveTo (hdc, parameter->x, parameter->y);
 LineTo (hdc, parameter->x+1, parameter->y);
}


/******************************************************************
Function:     Call the marker drawing function whose ID is stored in the ELEMENT
              node.
Parameters:   elementPtr -- pointer to the ELEMENT node.
******************************************************************/
void  Call_DrawMarker(struct ELEMENT *elementPtr)
{
 struct param_DrawMarker *parameter;

 parameter = (struct param_DrawMarker *)(elementPtr->paramPtr);
 drawMarker(parameter->x, parameter->y, parameter->markerSize,  \
                     parameter->markerStyle);
}
```

```
/****************************************************************
Function:     Call the line drawing function whose ID is stored in the ELEMENT node.
Parameters:   ptr -- pointer to the ELEMENT node.
****************************************************************/
void Call_lineCoord(struct ELEMENT *ptr)
{

 char chBuf[80];
 struct param_lineCoord *parameter;

 parameter = (struct param_lineCoord *)(ptr->paramPtr);
 MoveTo (hdc, parameter->x1,parameter->y1);
 LineTo (hdc, parameter->x2, parameter->y2);
}

/****************************************************************
Function:     Call the ellipse arc drawing function whose ID is stored in the
              ELEMENT node.
Parameters:   elementPtr -- pointer to the ELEMENT node.
****************************************************************/
void  Call_ellipseArc(struct ELEMENT *elementPtr)
{
 struct param_ellipse *parameter;

 parameter = (struct param_ellipse *)(elementPtr->paramPtr);
 Arc(hdc, parameter->left, parameter->top, parameter->right,  \
    parameter->bottom, parameter->x1, parameter->y1, parameter->x2, \
    parameter->y2);
}

/****************************************************************
Function:     Call the ellipse drawing function whose ID is stored in the
              ELEMENT node.
Parameters:   elementPtr -- pointer to the ELEMENT node.
****************************************************************/
void  Call_FilledEllipse(struct ELEMENT *elementPtr)
{
 struct param_ellipse *parameter;

 parameter = (struct param_ellipse *)(elementPtr->paramPtr);
 Pie(hdc, parameter->left, parameter->top, parameter->right,  \
    parameter->bottom, parameter->x1, parameter->y1, parameter->x2, \
    parameter->y2);
}
```

```
/*****************************************************************
Function:     Call the polyline drawing function whose ID is stored in the
              ELEMENT node.
Parameters:   ptr -- pointer to the ELEMENT node.
*****************************************************************/
void Call_polyLineCoord(struct ELEMENT *ptr)
{
 struct param_poly *parameter;
 int vertics;
 char chBuf[80];

 parameter = (struct param_poly *)(ptr->paramPtr);
 vertics = parameter->vertics,
 polyLineCoord(vertics, &(parameter->ptr[0]),&(parameter->ptr[vertics]));
}

/*****************************************************************
Function:     Call the polygon drawing function whose ID is stored in the
              ELEMENT node.
Parameters:   elementPtr -- pointer to the ELEMENT node.
*****************************************************************/
void  Call_Polygon(struct ELEMENT *elementPtr)
{
 struct param_poly *parameter;
 int vertics;
 char chBuf[80];

 parameter = (struct param_poly *)(elementPtr->paramPtr);
 vertics = parameter->vertics;
 polygon(vertics, &(parameter->ptr[0]),&(parameter->ptr[vertics]));
}

/*****************************************************************
Function:     Call the text string output function whose ID is stored in the
              ELEMENT node.
Parameters:   ptr -- pointer to the ELEMENT node.
*****************************************************************/
void  Call_text(struct ELEMENT *elementPtr)
{
 struct param_text *parameter;
 int stringSize;
 parameter = (struct param_text *)(elementPtr->paramPtr);
 stringSize = parameter->stringSize;
 TextOut(hdc, parameter->x, parameter->y, parameter->ptr, stringSize);
```

```
}
```

Appendix III

```
------------------------------------------------------------------------------------------------
-- Example application which uses GNA95GP storage structure, it is designed to
-- continuously rotate a triangle about the y-axis and display it.
------------------------------------------------------------------------------------------------
with init;
with objtype;
with Output;
with attribute;
with attrtype;
with structure;
with mymath;
procedure userMain is

use Output;
use objtype;
use attrtype;
use init;
use structure;
use attribute;
use attrtype;
use mymath;

p1, p2:      tagPoint;
X_list:      VErtexXCoordList(0..10);
Y_list:      VErtexYCoordList(0..10);
z_list:      array(integer range 0 .. 10) of integer;
X:           fixed;
Y:           fixed;
XX:          fixed;
YY:          fixed;
xxx:         integer;
yyy:         integer;
zzz:         integer;
X1:          integer;
Y1:          integer;
Z1:          integer;
x00, y00, z00:     integer;
vertics:     VErtexList(0..10);
rect:        tagRectangle;
```

```
start_angle, end_angle: angleType;
i:          integer;
ret:        integer;
--pragma suppress(Range_Check,  On =>XCoord);
pragma suppress(All_Checks);
begin
  Gnat95gp_init;
  if Gna95gp_openStructure(1)=1 then
  --setting color to be green.
  Gna95gp_setColor(10);                  --element #1
  X_list(0) := 300;
  X_list(1) := 300+80;
  X_list(2) := 300+80+80;
  X_list(3) := 300;


  Y_list(0) := 200;
  Y_list(1) := 200+100;
  Y_list(2) := 260;
  Y_list(3) := 200;

  for i in 0 .. 3 loop
    vertics(i) := makePoint(X_list(i), Y_list(i));
  end loop;
  Gna95gp_Polygon(4,  vertics);              --element #2, drawing the triangle.
  Gna95gp_closeStructure;
  end if;

  -- initilize the Z-arixs coordinates for the triangle.
  for i in 0 .. 3 loop
   z_list(i) := 0;
  end loop;
  while TRUE loop
  Gna95gp_waitkeyEvent;
  for i in 0 ..36 loop
    -- (x00,y00,z00): the original point this triangle rotate about.
    x00 := X_list(1);
   z00 := z_list(1);
    for j in 0 .. 2 loop
     X_list(j) :=  X_list(j) - x00;
     z_list(j) := z_list(j) - z00;
     X1 :=  X_list(j);
     Z1 := z_list(j);
     xxx := integer(CCos(X1, 10.0) + SSin(Z1, 10.0));
     zzz := integer(CCos(Z1, 10.0) - SSin(X1, 10.0));
```

```
    X_list(j) :=  xxx + x00;
   z_list(j) := zzz + z00;
   end loop;
   X_list(3):= X_list(0);
   Y_list(3):= Y_list(0);
    -- delete this structure.
   ret := Gna95gp_deleteStructure(1);

   -- create a new structure
   if Gna95gp_openStructure(1)=1 then
   -- set color
   Gna95gp_setColor(10);

   for j in 0 .. 3 loop
     vertics(j) := makePoint(X_list(j), Y_list(j));
   end loop;
   Gna95gp_Polygon(4,  vertics);
   Gna95gp_closeStructure;
   end if;
   Gna95gp_waitkeyEvent;
  end loop;
  end loop;

end userMain;
pragma Export(C, userMain, "UserMain");
```

VITA

Lan Li

Candidate for the Degree of

Master of Science

Thesis: DESIGN AND IMPLEMENTATION OF A 2D GRAPHICS PACKAGE IN ADA 95

Major Field: Computer Science

Biographical:

Education: Graduate from the Seventh School of ChenDu, SiChuan, China in June 1983; received Bachelor of Science in Computer Science Department from ZheJiang University, HangZhou, China in June,1987. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1996.

Experiences: Computer Programmer, China Research Institute of Printing Science and Technology, Beijing, China, August 1987 to May 1991; Computer software Engineer, R'International Systems CO., LTD, Tokyo, Japan, June 1991 to June 1994; Oklahoma State University, Department of Computer Science, August 1994 to present.