

OBJECT-ORIENTED DATA STRUCTURE
ANIMATION

By

SHRAVAN KUMAR ARRA

Bachelor of Technology

Jawaharlal Nehru Technological University

Hyderabad, India

1989

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
July, 1992

100-100000-1

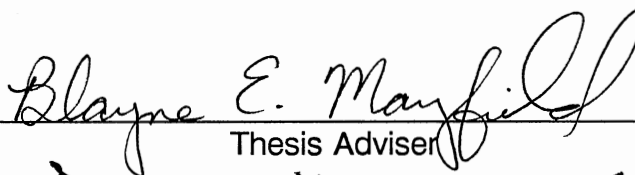
Thesis
1992
A7730

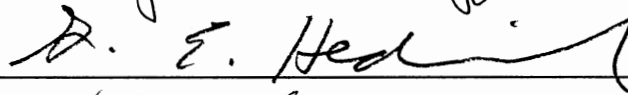
100-100000-1

Thesis
1992
A7730

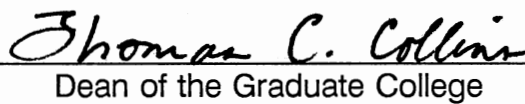
OBJECT-ORIENTED DATA STRUCTURE
ANIMATION

Thesis approved :


Thesis Adviser






Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express sincere appreciation to Dr. Blayne Mayfield for his encouragement, advice and moral support during the course of the thesis. Many thanks go to Dr. Miller for his valuable suggestions and help with relevant material. My thanks also go to Dr. Hedrick for his encouragement to participate in the seminar at Oklahoma Academy of Sciences. I thank Dr. Mayfield, Dr. Miller and Dr. Hedrick for serving on my committee.

I extend my sincere thanks to Dr. John Stasko of Georgia Institute of Technology for his valuable suggestions. His Xtango program was very useful in designing this application. I also thank Dr. Sharda for allowing me to use his computer hardware and software. My sincere thanks to Office of Business and Economic Research for their help with the 386 computer.

My parents and family supported me all the way and helped me keep the end goal constantly in sight. Thanks also go to my close friend Sriram who gave me moral support through out my thesis. I also thank my roommates Venu, Jasti and Karra.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. RELATED WORK	4
III. PROPOSED SYSTEM	9
User's View	9
Programmer's View	10
Program Component	11
Animation Component	12
IV. OBJECT-ORIENTED PROGRAMMING AND C++	14
Object-Oriented Concepts	14
Abstraction	15
Encapsulation	15
Modularity	16
Hierarchy	17
Polymorphism	18
V. DESIGN AND IMPLEMENTATION	20
Object-Oriented Analysis	20
Identification of Objects	21
Definition of Classes	22
Hierarchical Organization of Classes	23
Object-Oriented Design	23
Design of the Graphical User Interface	24
Design of Animation Library	26
Implementation	28
Implementation of Graphical User Interface	28
Implementation of Animation Library	31
VI. FUTURE DIRECTIONS	39
VII. CONCLUSIONS	40

BIBLIOGRAPHY 41

APPENDIXES

APPENDIX A - USER'S GUIDE 44

APPENDIX B - WINDOWS PROGRAMMING 51

LIST OF FIGURES

Figure	Page
1. Overview of the System	12
2. "is a" and "has a" Relationship	18
3. Example of a User Class	45
4. The Main Window of the GUI	47
5. The About Dialog Box	48
6. The OopaRect Class	50
7. The OopaArrow Class	50

CHAPTER I

INTRODUCTION

"A picture is worth a thousand words." Often one does not find bugs in a program though they exist. If the same program is presented in the form of pictures, it may be easier to find bugs. To understand any object, one almost always starts with pictures though one may not need them later. Program animation shows a lot of information in pictures, thus gives better understanding of programs and algorithms. A new and improved algorithm of dynamic Huffman trees came from viewing an animated version of Knuth's dynamic Huffman trees[JSV85]. Such software is also useful for instructors, project engineers etc. in explaining to their clients (students, programmers, etc.) what an algorithm does and what it is supposed to do. For example, a student may get a better understanding of Quicksort by viewing an animated version of it with different sets of input.

A program may have one or more data structures. To animate a program one must animate the data structures of a program. A data structure is the mathematical model of the data in an ADT (Abstract Data Type)¹, and it represents the ways in which data items of the ADT are related. In other words

¹An Abstract Data Type is a mathematical model with data and a collection of operations defined on that data.

a data structure consists of data and the operations that can be applied to that data. In object-oriented programming, data and operations are grouped together and called an object. When operations are performed on a data structure, the data may be changed and hence the physical structure of the data structure may also be changed. To a programmer dealing with textual results of a program, it takes a lot of time to understand the results that are presented in this form. Data structure animation aims at graphically displaying the changes in the data structure, using incremental motion. For example, a push operation on a stack data structure might be displayed by moving an element (a rectangle may be used to represent each element of the stack) to the top of the stack marked by an arrow. The arrow is then moved upwards to show the updating of the top pointer. Displaying a data structure at a point of time only gives the current state of the data structure. Though it is good to be able to see the current state of the data structure in the form of a picture, it may not help to understand what operations are being done and how the operation is being performed. This thesis aims at dynamically displaying the changes in the data structure as they are done. Such displays can lead to a better understanding of algorithms and data structures.

The first application of such a visualization system is computer science instruction. In courses dealing with algorithms and data structures this will be a very useful tool to demonstrate concepts. Students can experiment with the system by viewing animations for different sets of inputs. This can reveal some special cases that were not known to the student before. Advanced students

can use this tool to develop new data structures. Similarly, such a tool can be used in research to experiment with new data structures. Another important application of program animation is in software maintenance. Once a large piece of software is developed it is very difficult for a person to make modifications in the future. But if the software is animated, it should be easier to understand the software. Program animation could assist in teaching deaf students. Simulation is another potential area where program animation can be applied. For example, when such a system is used to simulate an operating system, one can see the graphical objects (representing the jobs) entering into the ready queue, moving to the CPU, etc. It also ought to be useful in program debugging.

Program animation can be static, where the animation can be seen only for a fixed set of inputs. It can also be dynamic, where the animation can be seen with different inputs. Animation of an object-oriented program consists of animation of its objects. In this thesis only dynamic program animation is considered. A prototype demonstrating the features and importance of such a package is also designed and implemented as part of the thesis.

CHAPTER II

RELATED WORK

Visualization as a means to understand programs and algorithms has been a subject of study since the 1960s. Visualization of programs in the early years was done more with video tapes and films. As the media suggest, the visualization was static, i.e if an algorithm was animated once for particular data it could be viewed any number of times only for those particular data. The disadvantage of such a system was that it did not allow users to experiment with the model being displayed. "L6: Bell Telephone Laboratories Low Level Linked List Language" produced by Knowlton[KK66] in 1966 was the first computer-generated movie. It showed how an assembly level list processing language works.

Hopgood's film on hashing algorithms(1974), Booth's[BK75] "PQ-trees"(1975), and Baecker's[RMB81] "Sorting out Sorting"(1981) are among other important films that were made. Hopgood's movie contained three views: a hash table, a graph indicating the number of probes needed to insert an item, and the maximum number of collisions to insert an item. The movie depicted the actions for different inputs. Booth's film showed the working of several algorithms on PQ-trees. When there was a change in the data structure, the transformation was shown in smooth incremental animation. Baecker's "Sorting

out Sorting" illustrated a number of different sorting algorithms.

Since the mid-1970s the focus of research in animation has changed to animation of data structures. Several systems have been built that automatically produce a static graphical display of a program's data structures from the information available to the system debugger at run time. The advantage of such systems is that data structures in any arbitrary program can be viewed without altering the program in any way. These systems give a static picture of the data structure. They do not show what operation is being performed and how the data structure is changing. They only show the picture of the data structure before and after the operation. Incense by Brad A. Myers[BAM83] followed this style. It is a prototype that lets the programmer view the data structures in a desired fashion. Incense was written in and for the Mesa programming language. The user specifies the variable name to get its pictorial display. One or more formats are associated with each data type. The user selects one of these formats. PROVIDE (Process Visualization and Debugging Environment) by Moher[TGM85] goes a step further into program debugging using graphical display of data structures. PROVIDE has the features of most of the traditional dynamic debuggers. It extends beyond these with additional features such as pictorial displays, interactive control, etc. It is also a prototype. It was actually introduced to demonstrate the principles behind a style of debugging which improves access to the details of program execution. It has been implemented on a multiple monitor workstation.

Research on algorithm animation systems started in the mid 1970s, but they

caught more attention only in the 1980s. Baecker's[RMB75] system developed in the mid-1970s was the first known system to aim at algorithm animation. Yarwood's[EY74] system for illustrating programs and DeBoer's[JMD74] system for animating micro - PL/I programs were among those developed at around the same time. BALSA-I and Animus were among those that came in the 1980s. BALSA(Brown University Algorithm Simulator and Animator) is a software environment designed to animate programs. The "algorithm designer" writes a software program to be animated and identifies "interesting events" in the programs. The "animator" implements the graphical pictures of the animation. The "script writer" uses the high level command facilities to produce scripts containing specific material for presentation to users. The "user" makes use of these scripts or directly interacts with the dynamic graphical representation of this algorithm. It was designed to work on a network interconnecting all the machines in a laboratory. On a network, the program can be run in broadcast mode where all the students can only view the algorithm, and in Independent mode where each student runs his own animation. London and Duisberg animated a collection of algorithms in Smalltalk[LD85] by using Smalltalk's MVC(Model-View-Controller) paradigm. Duisberg's Animus system used "temporal constraints" to describe the appearance and structure of a picture as well as their evolution over time. Brown's BALSA-I[MHB89] concentrated more on algorithm animation and introduced the concept of interesting events and scripts. BALSA-II was later developed from BALSA-I. Kleyn and Gingrich applied the animation techniques

to object-oriented systems in their system called GRAPHTRACE[MP88]. In this system they concentrated on presenting concurrently animated views of object-oriented systems. In this a viewer can see object-object relationships, a method invocation graph, etc. concurrently. The user first executes the program in recording mode. In this mode, information such as originator of a message, recipient object, name of the invoked method and values of arguments, is recorded and stored. Then the user runs the program in animate mode where the user views the above information in a graphical display. The speed of the animation can be varied. PECAN[SPR84] also presents concurrent views of a running program. PECAN presents multiple views of the user's program and its semantics. It presents a syntax-directed editor, a declaration editor, a structured flow graph editor, expression trees, flow graphs etc. This is primarily a program development system rather than a program animation system. An interesting feature of this system is the reverse execution display of a program. Currently this system cannot provide data views such as different graphical displays for trees, queues, etc. Thinkpad[RES85] and P.V.[GRCDP85] are other programs that are connected to this area. Stasko's TANGO[JTS90] is the latest among the algorithm animation systems. In this, Stasko introduces the path-transition paradigm for animation design. Tango is a framework that simplifies animation design. It works with three components; an algorithm component, a mapping component, and an animation component. The path transition paradigm is based on four abstract data types: locations, images, paths, and transitions. Present research claims to improve Tango to animate

object-oriented programs and parallel programs. A tool called Dance(Demonstrational ANimation CrEation) is also being designed.

Visual technology not only influenced the animation of programs but also the whole programming environment. Developments in graphics workstations are also aiding this process of visualization. The above research and the growing visual technology demonstrate the importance of current research in program visualization.

CHAPTER III

OVERVIEW OF THE SYSTEM

Description of the problem

The purpose of this package is to provide a prototype to demonstrate data structure animation. The prototype has a simple user-friendly Graphical User Interface(GUI) and a library that aids the animation of data structures. The library provides classes to animate basic data structures such as arrays, stacks, queues, and linked lists. The GUI should help the user in using the prototype. The user can run an animation, zoom in, zoom out, and change the speed of the animation.

The system can be viewed differently from a user's point of view and the programmer's point of view.

User's View

The animation of the user program is done in three steps. They are

- 1) The animator creates the animations.

Animations are created by writing animation classes using the animation class library. This library has classes for basic objects that are needed for animation. The graphical objects that are provided to him are rectangle, arrow, array, etc.

Some operations provided include movement, and change of color and size.

2) The user annotates the program.

The user opens the source program and marks those statements after which a change in the state of the animation is needed. The user also specifies which animation function is to be called when that statement is encountered.

3) The user executes the program and watches the animation.

During the execution of the program the user can toggle between two modes:

- a) View mode: In this mode the user can only view the animation of the program. Once the program starts execution the animation starts in the edit mode.
- b) Edit mode: In this mode the user is provided with the following options:
 - i) Change the speed of animation.
 - ii) Zoom in/zoom out a part of the animation window.
 - iii) Scroll the window to see the covered portions of the data structure.

Programmer's View

This system aims at animating object-oriented programs. With this a user can obtain a graphical picture of the data structures of a program. As operations are being performed on a data structure the changes in the data structure are shown as movements and transitions of graphical objects that represent the data structure. The transitions include changes in size, color, etc. Since this is a research project, only a prototype demonstrating the abilities is built rather than a full-fledged package. The most important point to be noted

in the development of such a system is that animation displays cannot be created automatically. An algorithm's operations cannot be deduced by the system from the source code but must be denoted by a person with knowledge of the operations performed by the algorithm[BS85]. Hence for each operation(a significant event) in the algorithm, the user creates the animation using the graphical objects that are provided to him in this package.

The system consists of the following two components :

- 1) Program Component
- 2) Animation Component

Figure 1 shows a block diagram of the system. The programmer writes a data structure program that needs to be animated. The animator(may be the programmer) designs the animation scenes using the animation component.

Program Component

The program component consists of a C++ program that has one or more data structures in it. This may be written with classes and other object-oriented techniques such as inheritance and overloading.

The user marks the interesting events. Interesting events are those where a change in the state of the animated version of the data structure is desired. For example, the change can be a change of the height of an object representing the change in the value of the object. When an interesting event is encountered, a message indicating the change desired in the animated data structure, is passed. Examples of interesting events are updating the TOP

pointer of a stack data structure, inserting a node in a linked list, balancing action in AVL-trees, etc. In a program using inheritance, it is possible to use inheritance in the animation program corresponding to the inheritance in the program being animated. For example, if an AVL tree inherits the three traversal methods(preorder, inorder, postorder) from a BS tree, then the animation class of the AVL-trees can inherit from the animation class of BS trees. In this case, one needs to take care to include all the classes in the files of the animation component.

Animation Component

The animation component contains a conventional program library of classes that are useful to the user in building the animations. The animator can use the library to write animation classes with greater flexibility. In the design of this library, a bottom-up approach is adopted starting with basic objects. The basic objects to be provided, such as rectangle, arrow, point, straight line, etc. are identified. The classes are then designed with data and operations. The program for this thesis also demonstrates the design of animations, using these graphical objects, of standard data structures such as structures, arrays, stacks, queues, linked lists, etc.

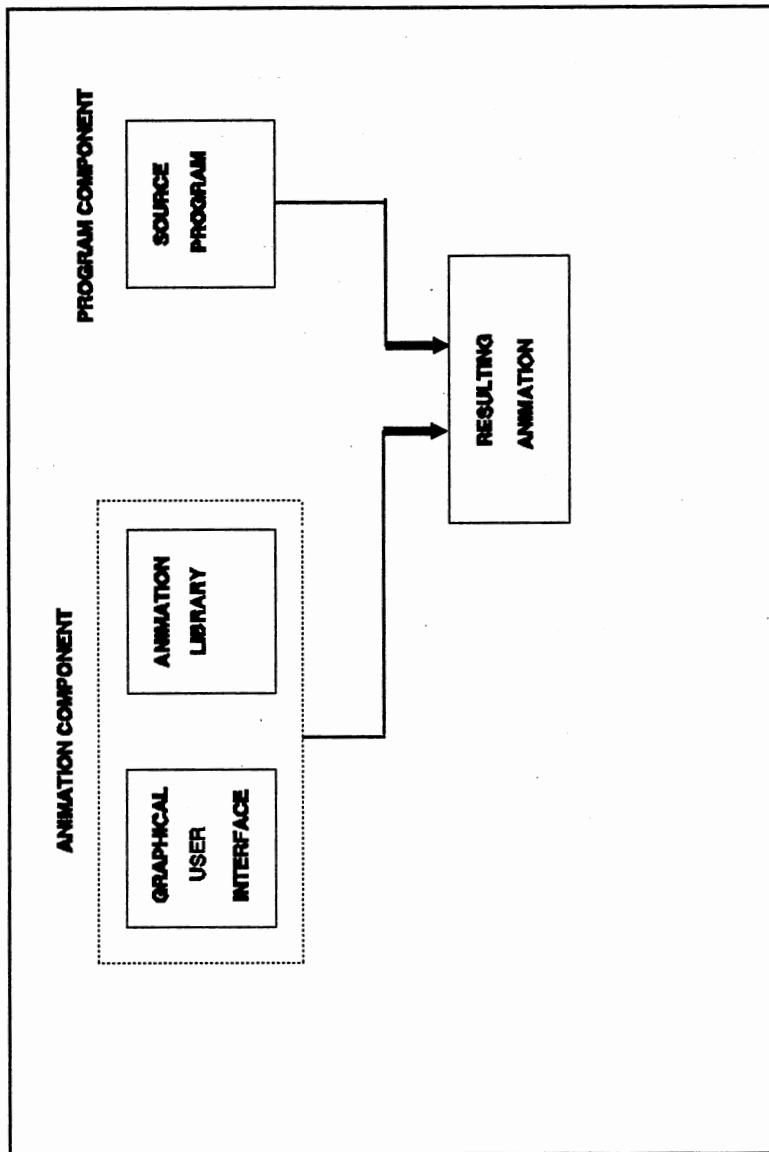


Figure 1. Overview of the System

CHAPTER IV

OBJECT-ORIENTED PROGRAMMING AND C++

Object-Oriented Concepts

Booch[BG91] describes object-oriented programming as "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united via inheritance relationships." In programming, a big problem is generally decomposed into smaller parts. In the traditional procedural approach, a problem is decomposed into smaller routines whereas in the object-oriented approach, a problem is decomposed into smaller objects. This helps in organizing the inherent complexity of software systems. Procedural programming techniques concentrate on functions and actions whereas object-oriented techniques concentrate on logical objects that contain both data and functionality in a single unit. The term "object-oriented programming" is derived from the object concept in the Simula 67 programming language.

Large software systems are generally very complex. It is often required to enhance or modify some part of the software. With software that does not follow abstraction and encapsulation principles(as described below), there is a

good chance that a lot of work would be needed to do the required modification. Hence it is very important that a software designer keeps abstraction and encapsulation in mind while designing the components. Object-oriented programs are implemented with abstraction and encapsulation. According to Booch[BG91], an object-oriented program must have the following properties :

- 1) Abstraction
- 2) Encapsulation
- 3) Modularity
- 4) Hierarchy

When implemented properly, these result in a system that is flexible and reusable. Such a system will have a long life.

Abstraction

An abstraction denotes only the essential characteristics of an object that the user needs to know. It focuses on the interface of the object and hence serves to separate the external interface from its internal implementation. C++ facilitates abstraction by means of public functions of a class. Public functions of a class are the only way an operation can be done on the private data of an object.

Encapsulation

Encapsulation, also known as data hiding, is a technique for minimizing

interdependencies among separately written modules by defining strict external interfaces. There should be no other way to manipulate an object's internal data. Encapsulation assures designers that compatible changes can be made safely without affecting the clients or users of the object. In C++, member variables and functions of a class can have one of the three attributes: private, protected, and public. A public variable or function is accessible both inside and outside the class. Protected variables can be accessed only by member functions of the class and member functions of its immediate child. Private variables can be accessed only within the class and are completely invisible outside that class. Hence by using the private construct data can be hidden or encapsulated.

Modularity

Booch[BG91] defines modularity as "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Modularization of a program consists of dividing a program into modules that can be compiled separately but which have connections with other modules. In traditional procedural programming, modularization is primarily concerned with the meaningful grouping of subprograms using the criteria of coupling and cohesion. In object-oriented programming, the problem is subtly different; the task is to decide where to physically package the classes and the objects from the design's logical structure which are distinctly different from subprograms[GB91]."

Hierarchy

A set of abstractions often forms a hierarchy and, by identifying these hierarchies the understanding of the problem is simplified. Booch states hierarchy as "a ranking or ordering of abstractions". There are two kinds of hierarchies: "is a" relationship and "has a" relationship. Inheritance falls under the "is a" category of hierarchy. Reusability is often an essential attribute of software. Inheritance is a reusability mechanism in object-oriented languages for sharing behaviors among or between classes. There are two types of inheritance: single inheritance and multiple inheritance. In single inheritance, the child class has only one parent whereas in multiple inheritance the child class has multiple parents. In C++ both "is a" and "has a" relationships can be used. An instance of a class can be defined as one of the private variables of another class. For example, the declaration of class B is as follows :

```
class B {  
    A anInst;  
    public:  
    ....  
}
```

Here anInst is an instance of class A. Class B contains an instance of class A. This is an example of the "has a" relationship. Inheritance can be used to get the "is a" relationship. The declaration of class B is as follows :


```
class B : public A {  
    .....  
    public:  
    ..... }  
}
```

In the following figure each oval represents a class.

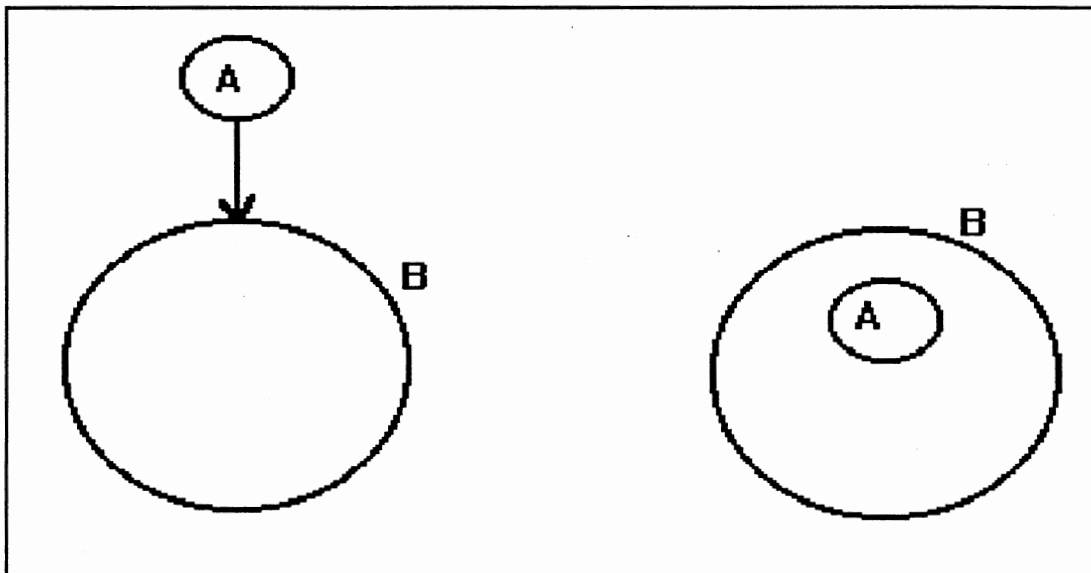


Figure 2. "is a" and "has a" Relationship

Polymorphism

Another important aspect of object-oriented programming is polymorphism. Polymorphism refers to an object's capability to select the correct internal method based on the type of data received in a message. This is also referred to as overloading. In C++, the overload keyword can be used to overload a function. There are two different types of overloading: function overloading and

given the same name as long as their arguments differ in their type. Sometimes a function needs to be written for different types of input. Function overloading helps to improve the interface provided to the users of the class. Sometimes an abstract class is defined from which other classes are derived. An abstract class cannot have instances of its own. It provides a common interface to its derived classes. The functions in the abstract class are made "pure virtual" functions by assigning the definition of that virtual function to zero. Since these pure virtual functions are redefined in the derived classes it is called pure polymorphism.

In operator overloading, a function is written with the name of the function as the operator. This is used mainly to make an operator accept arguments of different type other than the standard type that is provided in the definition of the language. For example, the operator "+" can be overloaded to add two matrices. In such a case, a function whose name is "+" is written with the matrices as arguments.

CHAPTER V

DESIGN AND IMPLEMENTATION

Object-Oriented Analysis

This system is an object-oriented system. Hence an object-oriented analysis is used. Object-oriented analysis is an approach that emphasizes defining the characteristics and behavior of the program within a system of objects. The purpose of analysis is to provide a detailed description of the problem. In object-oriented analysis, we seek to model the world by identifying the objects that form the vocabulary of the given problem. Then the commonalities between the objects are observed and the objects are grouped. A common definition for each group can be given in the form of a class. Once the classes are identified, logical relationships and interactions among them are noted and a hierarchy of classes is obtained. At the end of the analysis, classes and their hierarchies are ready for design. To support object-oriented programming the following points are focussed :

- 1) Identification of objects and definition of classes
- 2) Hierarchical organization of classes

For the convenience of the reader the description of the problem is reproduced in this chapter. The purpose of this package is to provide a

prototype to demonstrate data structure animation. The prototype has a simple user-friendly Graphical User Interface(GUI), and a library that aids data structure animation. The library provides classes to animate basic data structures such as arrays, stacks, queues, and linked lists. The GUI helps the user in using the prototype. The user can run an animation and change the speed of the animation.

Identification of Objects

The above problem states that the prototype has a simple user friendly GUI. It allows the user to run an animation and change the speed of the animation. When the user has the option to do more than one action at a given point of time, a menu is desirable. Since the animation is presented in a window, which can show a data structure only partially, a way to scroll is needed. The user should be able to zoom in or zoom out if needed. Winblad[WLA88] suggests that the objects can be identified by picking the nouns from the problem description and their interface from the verbs. In the nouns of the above discussion one can see menu and window. The actions or verbs are, run, change speed, scroll, zoom in, and zoom out. So, a menu to select the above actions is needed. A window to present the animation is also needed and it should be scrollable.

The problem also states that, "The prototype has an animation library that aids data structure animation". The data structures include arrays, stacks, queues, and linked lists. An array has a fixed number of elements. All the

elements are of the same type and their values can vary. Translating this into the animation domain, an array has fixed number of graphical objects. All the graphical objects of the array are similar conceptually but may vary in magnitude and quality. A stack is a list of similar items. Only the item at the top can be taken out (popping). A new item can be inserted only at the top (pushing). A queue is a list of elements. Only the front item in the queue can be taken out. A new element can be added only at the tail of the queue. Similarly a linked list is a list of elements in which each element points to the next element. From the above discussion it can be concluded that every data structure has elements in it. In the animation domain the element is represented by a graphical object. Pointers are needed to represent how each element points to the next element in a list. Arrows can be used to represent pointers.

Definition of Classes

After the objects are identified classes are listed. In this problem, the following classes can be listed.

- 1) Class Menu
- 2) Class Window
- 3) Class Graphical object
- 4) Class Arrow
- 5) Class Array
- 6) Class Stack

7) Class Queue

8) Class List

Hierarchical Organization of Classes

From the previous description, it is clear that the GUI and the animation library can be treated independently. In the GUI part, there are only two classes. In Microsoft Windows there are two types of windows: windows with a menu and windows without a menu. Here a window with a menu can be used. In the animation library section, each of the classes array, stack, queue, and linked list have graphical objects inside them. So a "has a" relationship can be established here. A stack can be implemented as a restricted list. A linked list is an improvement over the stack. So a linked list can be derived from the stack. Here "is a" relationship can be used.

With classes identified and hierarchy established the system is ready for design.

Object-Oriented Design

Object-oriented design starts as soon as a formal or informal model of the problem is ready. It concentrates on identifying the semantics of the classes identified above. The functions or capabilities of each class are listed. These functions constitute the interface or protocol of a class. Designing the interface for a given object may require decisions that change the design of another object.

Design of the Graphical User Interface

In the previous section, a window with a menu was identified as one class needed for the graphical user interface. The actions that can be performed on this window are display, scroll, paint, and selection of menu items. Since Microsoft Windows is inherently object-oriented in part, selection of menu items is already implemented. The interface to this window class is thus only scrolling, painting, and displaying. During the process of implementation, more functions may be added. Functions such as opening, closing, and moving are inherent in a window created using Microsoft Windows.

The Show function displays the window. This function must be called after registering the window.

The PaintWindow function is the gateway to the window. Anything that is to be painted on the window has to go through this function. This enables recording all the paint commands that paint on the window. Every time an object is to be painted on the screen, a PaintWindow message is sent to the MainWindow class. This includes information about where to paint and what to paint. All this information is stored in a node and inserted in a linked list. An id number is given to each object painted in the window. When an object is to be removed from a window, a delete message is sent to the window along with the id number of the object to be deleted. The PaintWindow routine deletes the object from the window and deletes its corresponding node from the linked list. Thus the linked list has only the nodes of the objects that are currently drawn in the drawing area. The drawing area is virtually an infinite area. Only a part of

this is visible at a given time. The window width may be only 640 pixels, but an object can be drawn anywhere from -x to x, where x is limited only by the type(eg., short int, long int, etc.) of x. The linked list is useful to store the information about all the graphic objects including those that are not visible on the screen. Initially the size of the drawing area is the same as that of a single full-size window. Whenever an object is to be drawn beyond this area, the drawing area is extended to include this new object in multiples of 600 pixels. Thus, if an object is to be added at (-500,200), the drawing area is extended to the left by 600 pixels.

The PaintWindow function goes through the linked list and draws all the objects that fall in the active window. Initially, the active window starts at (0,0). But when a user scrolls, the active window changes and the PaintWindow function is called to repaint the window to reflect the new active window. The PaintWindow function translates the logical co-ordinates to the real co-ordinates using the following formulae :

$$X_{\text{real}} = X_{\text{logical}} - X_{\text{active}}$$

$$Y_{\text{real}} = Y_{\text{logical}} - Y_{\text{active}}$$

It draws each object that is in the active window according to the new real co-ordinates it calculates. Whenever the drawing area grows in size, the scrollbar range and the position of the scrollbar thumb change to reflect the new position. The user can click on the left or right arrow to scroll to the right or left by 20 pixels respectively. The user can click on the scrollbar to the left or right of the thumb to move half a page (300 pixels) right or left. Similarly the vertical

scrollbar can be used to move up and down.

The zoom in and zoom out functions are built into the PaintWindow function. The PaintWindow function scales the images according to the current ZoomScale. The value of ZoomScale is 1 for a normal view, 0.5 for zoom in and 4 for zoom out view. The active window coordinates are also recalculated according to the ZoomScale.

Design of Animation Library

In the analysis section, the following classes were identified : GraphicObject, Arrow, Stack, Queue, and Linked List. In this section the interface to each of these classes is designed.

GraphicObject Class : Draw, move vertically, move horizontally, change width, change height and change color are the operations that can be done on an instance of this class. The object is drawn on a bitmap and a message is sent to the MainWindow class to display this bitmap. Information about the location, size and color of the bitmap is also sent.

Arrow Class : Displaying and erasing an arrow are the main functions of this class.

Stack Class : The operations that can be done on a stack are push and pop. When a push operation is done, a rectangle (or a circle) is pushed onto the top of the stack. The top pointer arrow moves up to display the increment in top. When a pop operation is performed the graphical object on top of the stack moves out and is erased. The top pointer moves down to indicate the

decrement in top.

Queue Class : The operations that can be done on a queue are enqueue and dequeue. When an enqueue operation is done, a rectangle (or a circle) moves to the end of the queue and the rear pointer is updated to show the new rear of the queue. When a dequeue operation is done, the rectangle (or a circle) at the front of the queue moves out and all the rectangles in the queue move one element to the front. The rear pointer is updated to show the new rear.

Array Class : The operations that are provided on the objects of this class are change the value of an element and swap two elements. When the value of an element changes, this is reflected by the change in the height of that element. The swap operation is implemented by moving the two elements, in an incremental motion, to their new position.

List Class : Inserting a node, deleting a node, and traversing the list are the main operations that can be performed on a list of this class.

Once the design is over, the problem is ready to be implemented. The advantages of object-oriented design are several. The implementation and design processes merge close together because object-oriented systems contain a great deal of design information. A good initial design can be expressed as an early version of the code. An object-oriented design also usually results in more reusable code than the process intensive procedural decomposition approach. It results in systems that are resilient to future changes. When change is necessary, the unique property of inheritance allows reuse and extension of the existing model. It is easy to have several teams

working independently on different parts of the design because the data and operations are localized within objects.

Implementation

The implementation process may give rise to change in the design of the problem. The problem is then redesigned before implementing. This prototype is implemented in C++ using Microsoft Windows.

Implementation of Graphical User Interface(GUI)

The GUI consists of a window with a menu. Each window also has a handle. Every window has a device context that describes the attributes of the window such as background color, foreground color, etc. There should be a way to obtain the handle of the window and the device context of the window in order to draw on it. Hence functions GetHandle and GetWinDC are provided. Every window should be capable of displaying itself on the screen. This function is provided by using the function Show. Apart from these there may be other attributes and operations that can be done on a window. Hence there is a basic window class and specific window classes are derived from this class. In Microsoft Windows, each window has a callback function generally called WndProc associated with it. This is the function which is responsible for doing the actions that a user requests. For example, if a user resizes a window, this function takes care of drawing the new window. This function varies from window to window, so it is not implemented in this class. Instead, it is declared

as a pure virtual function. This makes the class an abstract class and hence no instances of this class can be created. The window needed for the GUI has to be registered in order to show it on the screen. This is done in different ways for different windows. So a `MainWindow` class is derived from the window class and improved by adding a window title and functions `Register` and `MainWindow`. The `MainWindow` function is a constructor that creates the window. The `WndProc` function is redefined in this class. It is responsible for many actions that the user performs on the window. It processes a user's action such as clicking in the window, selecting a menu item, quitting from the application, etc. With these considerations a basic window class and a `MainWindow` class are implemented.

The menu is a resource that can be called when registering the window. It is prepared using Borland's WRT(Whitewater Resource Toolkit). This menu is given a resource id and used when registering the window class. Each menu item is given a different id number. When a user selects a menu item, a `WM_COMMAND` message is sent to the `WndProc` function with the id number of the menu item as the `wParam` parameter. The `WndProc` tests the `wParam` in a big switch statement and performs the corresponding action.

The GUI also consists of three dialog boxes: about box, speed box, and open box. When the user clicks ABOUT under the SPECIAL menu a dialog box showing the application name, its icon, and the author's name, comes up. This icon is actually a bitmap. The SPEED option under the OPTIONS menu displays a dialog box with a scrollbar control and buttons. The user can click

to the left or right to decrease or increase the speed. The dialog boxes process input from the user and perform some actions. So a callback function is written for each dialog box. Whenever a dialog box has to be displayed, an instance of this procedure is created and the control(user input) gets transferred to the dialog box. Now all the mouse and keyboard messages are received by the dialog box callback function. When the user closes the dialog box, the control gets transferred back to the WndProc. The dialog boxes are also predesigned resources. They are also designed using Borland's WRT.

The application is minimized into an icon when the user clicks on the minimize button on the top right corner. The application opens up when double clicked on this icon. Windows sends a WM_PAINT message to the WndProc. Initially the window is blank when maximized. So the application has to paint when it receives a WM_PAINT message. In this case, it paints the active window contents. Resizing also erases the window. But a WM_PAINT message is sent when the window is resized or when it is focussed. The code under the WM_PAINT case in the WndProc takes care of painting the active window. Selecting ZOOM IN option under the VIEW menu, makes the image bigger. The ZOOM OUT option under the VIEW menu gives a smaller image on the screen. The user selects the NORMAL option under the VIEW menu to come back to the normal view. This is implemented by erasing the screen and redrawing the screen with the new scaled images and co-ordinates. The ZOOM IN option is not currently supported for Super VGA monitors. All the resources are released when the user selects QUIT under the FILE menu.

Up/down scrolling and left/right scrolling is also implemented. A `WM_HSCROLL` message is sent to the `WndProc` when the user clicks on the horizontal scroll bar. At this point, the `wParam` parameter is checked to determine if the user clicked on the left arrow or the right arrow or on the left side of the caret or on the right side of the caret. The caret is a small square in the scroll bar that indicates the current position of the image inside the window. It is also referred to as the thumbtrack. Depending on the value of the `wParam` parameter, the corresponding action is carried out. Clicking on the right arrow scrolls the contents of the window 20 pixels to the left. The position of the caret is updated. The uncovered portion of the window is painted by sending a `PaintWindow` message to the window. Similarly vertical scrolling is implemented. Caret dragging is not implemented.

Implementation of Animation Library

This involves the implementation of the classes that were identified before.

GraphicObject Class : A graphical object can have current position, height, width, and color information as its attributes. It can have other attributes that help in implementing the class such as id number and the bitmap. The methods that were identified are :

DrawRect : A graphical object is drawn when this message is sent to the object. To draw a graphic object one needs to specify the position, height, width, color, and shape of the object. If a text is to be displayed, the text should also be specified. This function draws a bitmap of the given shape. The object cannot

be more than 100 pixels wide and 100 pixels tall. If a width or height of more than 100 pixels is specified, a width or height of 100 pixels is taken. The color is specified by sending a Color structure. The Color structure consists of three fields: red, blue, and green. These fields are integers and have a value in the range 0 to 255. When this message is sent to the GraphicObject class, it sends a message to the window class to paint in the drawing area. Thus all painting occurs in the window class. This function draws the object on a bitmap and this bitmap is sent to the window class.

MoveRectHoriz : The graphical object should be able to move horizontally. To move the graphical object, the distance to be moved is to be specified. If the distance is negative, the object moves to the left, and if it is positive it moves to right.

MoveRectVert : To move the graphical object vertically, the distance to be moved is to be specified. If the distance is negative, the object moves up and if it is positive, it moves down.

A rudimentary way of achieving the moving effect is to erase the object in the current position and redrawing it in a new position. If the new position is in the direction of motion, and, if this redrawing process is continuously done the object moves. But when this approach was taken, there was a lot of flicker in the image. Another approach is to have a border of the same color as the background, around the object. This can be done in two ways. Consider a rectangle object whose height is 'h'. One way is to draw a rectangle with h-2 pixels height and 1 pixel of border on both top and bottom. This approach

leads to a smaller rectangle than requested. It is desirable to have a border of two or more pixels. In this case, there is a big difference between the actual height specified and the height of the drawn rectangle. Instead, a border can be drawn outside the rectangle. This makes the actual height of the rectangle $h+2$. This may lead to problems if the user is not aware of the implementation. For example, if a user draws rectangles of height 10, one at $y=30$ and one at $y=40$, the border is lost on the common boundary of the two rectangles. So a dynamic approach is adapted which will not only avoid the flicker but also will give the exact width and height as specified.

In this approach, the object is drawn as usual. When the object is to be moved, a bitmap of size 5 pixels wider or taller is created depending on the direction of motion. A white border is drawn on only one side of the bitmap depending on the direction of movement. For example, for a vertical upward motion, a white border is drawn at the bottom edge of the bitmap. The actual object is drawn on the bitmap in the rest of the area. When this bitmap is displayed on the screen in incremental positions, the objects move smoothly without flicker.

A delay is introduced between two steps. This delay depends on the current speed as selected through the SPEED option under the OPTIONS menu. The delay is device independent.

DeleteRect : This method erases the graphical object from the screen. It does this by painting the area occupied by the object with the color of the background. It does this by sending a message to the MainWindow class. The

corresponding node is deleted from the linked list. The actual object is not deleted, but is only erased from the screen.

ChangeRectWidth, ChangeRectHeight : To change the height or width of the object, specify the new width or height of the object.

This is implemented by deleting the current object and redrawing it with the new width, or height.

OopaArrow Class : An arrow can have the current position, direction, width and height as the attributes. These are declared as private variables. The methods that were identified are :

DrawArrow : This method draws the arrow at the specified location. The width, height, and direction of the arrow are to be specified. The arrow cannot be wider or taller than 100 pixels wide. If a width or height of more than 100 pixels is given, a width or height of 100 pixels is assumed. The method actually draws an arrow bitmap already created using Borland's WRT. It loads the bitmap from the resource file that was already created and draws it in a temporary bitmap. This bitmap is sent as a parameter in the message sent to the window class.

DeleteArrow : This erases the given arrow from the screen. It does this by painting the area occupied by the object with the color of the background. It does this by sending a message to the MainWindow Class. The corresponding node is deleted from the linked list. The actual object is not deleted but is only erased from the screen.

Array Class : This class represents an array of graphical objects. The methods that were identified are:

DrawArray : This method draws an array of graphical objects of fixed size. This method is overloaded to provide a default draw method. In one method, the user does not need to specify anything. In another, the user needs to specify the position of the first element of the array, width, height, shape, color of the elements of the array, and distance between the elements.

DrawVarArray : This method draws an array of variable sized elements, the height of the element being proportional to the value it represents.

The array class consists of an array of graphic objects. A DrawRect message is sent to each of the elements in the array.

Swap : This method swaps two specified elements. This is done by exchanging the position of the objects. If object1 and object2 are being swapped, object1 moves slowly up and then to the right. Object2 moves slowly down and then to the left. Object1 moves down to take place of object2 and vice versa. This effect is achieved by sending MoveRectHoriz and MoveRectVert messages to the objects being swapped.

ChangeElementHeight : This method changes the height of the specified element of the array. This represents the change of the value of an element.

ChangeElementWidth : This method changes the height of the specified element of the array. This represents the change of the value of an element.

These two methods are implemented by deleting the old element and drawing the element with the new width or height.

Stack Class : This class represents a stack of graphical objects. The methods that were identified are :

OopaStack : This is the default constructor for the stack class. This is overloaded to provide a more customized constructor where the position of the stack, width, height, shape, and color of each node can be specified. This class is used to animate a stack implemented using arrays. Thus an array of graphical objects is created in the constructor.

~OopaStack : This is the destructor which deallocates the memory that was allocated for the array of graphical objects.

OopaPush : This function moves a node from outside the stack onto to the top of the stack. The object is initially drawn to the left of the first element of the stack. This object moves up by the size of the sack, then it moves horizontally to the maximum position of the stack and then down to the current top of the stack. It updates the arrow position to reflect the new stack top.

OopaPop : This function moves a node from the top of the stack vertically to the maximum height of the stack and then it moves horizontally. The object moves down and is erased finally. It also updates the arrow position to reflect the new stack top.

Queue Class : This class represents a queue of graphical objects. The methods that were identified are :

OopaQueue : This is the default constructor for the queue class. This is overloaded to provide a more customized constructor where the position of the queue, width, height, shape, and color of each node in the queue can be specified. This class is used to animate a queue implemented using arrays. Thus an array of graphical objects is created in the constructor.

`~OopaQueue` : This is the destructor which deallocates memory that was allocated for the array of graphical objects.

`OopaEnqueue` : This method moves a node from outside the queue to the rear of the queue. It updates the position of the rear arrow.

`OopaDequeue` : This method moves a node from the front of the queue and deletes it. It moves each node in the queue by one element towards the front and updates the position of the rear arrow.

`LinkedList` class : This class represents a linked list of graphical objects. The methods that were identified are :

`OopaList` : This is the constructor for the list class. It is overloaded to provide a customized constructor where the user can specify the location of the list, width, height, color, and shape of the nodes. Each node in the linked list has a graphical object and an arrow object.

`~OopaList` : This is the destructor for the list class. It deletes each node from the head until there are no more nodes. It finally deletes the header.

`CreateList` : This method draws the header of the list. It is called before any insertions are made.

`InsertInList` : This method inserts a node in the list. This moves a new node from just above the header of the list until it reaches the correct position. It then redraws the list with this node inserted in it.

`RedrawList` : This method erases the graphical objects that are part of the list and redraws the list.

`AppendToList` : This method appends a node to the list. This moves the new

node from outside the list until it reaches the end of the list.

DeleteNode : This method searches for a given node and erases the node it found. It redraws the list after deletion.

CHAPTER VI

FUTURE DIRECTIONS

This package is very useful in many ways. But to design animations, a programmer has to write code for the animation using the animation library. This is a time consuming process. Unless the animation is going to be used for a long time, or by many people, it may not be an easy tool to use. So we need an easier way to build animations. One way is to build a visual programming kit specifically designed for use with this tool. Such a kit should enable the animator to build animations by simple demonstrations. The demonstrations are sensed by a program and code consisting of calls to the library routines is generated. For example, the user should be able to create graphical objects, move them, and resize them, using just a mouse. Thus the user can build animations by pointing and clicking instead of writing code using animation library routines. An extensive animation library with more graphical objects and operations will help build visually attractive animations. If the time taken to build animations is reduced, the application can be used more efficiently in debugging. Since the animation library is built using object-oriented techniques it is easier to add more graphical objects using existing objects. Composite objects can be designed using Multiple Inheritance. If curved paths are added to the existing library, the package can be used to visualize the motion of particles.

CHAPTER VII

CONCLUSIONS

Day by day programming is becoming more efficient. Better methods to write programs are being sought as the need for efficient maintenance is increasing. Computers have become a part of almost everybody's life. A day may come when everybody will know computer programming. The concepts of object-oriented programming helps in efficient maintenance. The program animation systems will be useful to make programs easily understandable. The growing visual technology and the research in this direction prove that learning programming is going to be very useful. The principles of program animation are also useful for visual debuggers. With these in mind, a prototype was built to demonstrate the animation of data structures such as arrays, linked lists, stacks, and queues. The technique of object-oriented programming has helped in providing access to the library so that users can derive from existing classes and build better animations without having to rewrite the whole class. The prototype was tested on a VGA and super VGA monitors. Only the zoom in feature does not work on super VGA monitors. The user needs Microsoft Windows 3.0 or later to run this package. It can be run in both standard and enhanced modes of windows.

BIBLIOGRAPHY

- [AM89] Ambler, Allen L. and Burnett, Margaret M. Influence of visual technology on the evolution of language environments. *IEEE Computer* 22, 2(October-1989), 9-22.
- [BAM83] Myers, Brad A. INCENSE: A System for displaying data structures. *Computer Graphics* 17, 3(July-1983), 115-125.
- [BK75] Booth, K. *PQ-Trees*, 16mm color silent film, 12 minutes, 1975.
- [BS85] Brown, Marc H. and Sedgewick, Robert. Techniques for algorithm animation. *IEEE Software* 2, 1(January-1985), 28-39.
- [EY74] Yarwood, Edward. Towards program illustration. M.Sc. Thesis, Department of Computer Science, University of Toronto, Toronto, ON, 1974.
- [GRCDP85] Brown, Gretchen P., Carling, Richard T., Herot, Christopher, F., Kramlich, David A., and Souza, Paul. Program visualization: Graphical support for software development. *IEEE Computer* 18, 2(August-1985), 27-35.
- [JMD74] James, DeBoer M. A system for the animation of micro-PL/I programs. M.Sc. thesis, Department of Computer Science, University of Toronto, ON, 1974.
- [JSV85] Jeffrey, Vitter S. Design and analysis of dynamic Huffman coding. *Proc. 26th Annual Symp. on the Foundations of Computer Science*, October 1985, pp. 293-302.
- [JTS90] Stasko, John T. A practical animation language for software development. *Proc. of IEEE 1990 Int'l Conf. on Computer Languages*. August 1990, pp. 1-10.
- [JTS90] Stasko, John T. Tango: A framework and system for algorithm animation. *IEEE Computer* 23, 2(September-1990), 27-39.
- [KK66] Knowlton, Kenneth C. L6: Bell telephone laboratories low-level linked list language, two black and white sound films, 1966.

- [LD85] London, Ralph L. and Duisberg, Robert A. Animating programs using Smalltalk. *IEEE Computer* 18, 2(August-1985), 61-71.
- [MHB89] Brown, Marc H. *Algorithm animation*. MIT press, Cambridge, MA., 1988.
- [MP88] Kleyn, Michael F. and Gingrich, Paul C. GraphTrace - Understanding object-oriented systems using concurrently animated views. *OOPSLA' 88 Conference*(September 25-30, San Diego, CA). ACM/SIGPLAN, New York, 1988, pp. 191-205.
- [RES85] Rubin, Rober V., Colin, Eric J., and Reiss, Steven P. ThinkPad: A graphical system for programming by demonstration. *IEEE Software*. 2, 1(March-1985), 73-78.
- [RMB75] Ronald, Baecker M. Two systems which produce animated representations of the execution of computer programs. *ACM SIGCSE Bulletin* 7, 1(February-1975), 158-167.
- [RMB83] Ronald, Baeker M. *Sorting out Sorting*. 16mm color sound film, 25 minutes. 1981.
- [SPR84] Reiss, Steven P. Graphical program development with PECAN program development systems. *ACM transactions on Software Engineering* 14, 6(June-1988) 849.
- [STY87] Isoda, Sadahiro., Shimomura, Takao., and Ono, Yuji. VIPS: A Visual debugger. *IEEE Software* 4, 1(May-1987), 8-19.
- [TGM85] Thomas, Moher G. PROVIDE: A Process Visualization and Debugging Environment. Technical Report, University of Illinois at Chicago, Chicago, IL, July 1985.
- [UR90] Urlocker, Zack .Object-oriented programming for Windows. *BYTE* 15, 2(May - 1990), 287-294.
- [WL84] Finzer, William. and Gould, Laura. Programming by rehearsal. *BYTE* 9, 2(June-1984), 187-210.
- [WLA88] Winblad, Ann L. *Object-Oriented Software*. Addison-Wesley, Reading, MA., 1990.
- [VR90] Haarslev, Volker and Moller, Ralf. A framework for visualizing object-oriented systems. *Proc. of ECOOP/OOPSLA '90 Conference*(Oct. 21-25, Ottawa, Canada). ACM/SIGPLAN, New York, 1990, pp. 237-244.

APPENDIXES

APPENDIX A

USER'S GUIDE

This chapter discusses how to use the animation library in a program to be animated. It also provides information on how to use the graphical user interface during animation.

This package can be used to animate a C++ program. To animate a program, one may use the animation classes provided in the animation library or write one's own animation classes. First, the file "OODSA.H" must be included in the program. The animation library in this prototype has classes to animate arrays, stacks, queues, and linked lists. To write more animation classes one can inherit from these classes or write completely independent classes. One can also create customized views of a given class by inheriting from that class. Since this is a prototype, the package can animate only a single instance of a data structure at a time. If the user tries to animate more than one instance of a data structure or multiple data structures, the user needs to take care of the position of each of these data structures on the graphics screen; otherwise, they may overlap.

In C++, a data structure can be written in the form of a class with data as private variables and operations as public members functions. The purpose of this package is to animate the operations done on the data structure, i.e, when an operation is done in the user program, it has to be shown graphically on the

screen. To achieve this effect, the user has to declare an instance of the animation class to be used, in the private section of the class declaration. The animation class can be one of the classes provided in the animation library or an animation class written by the user. For example, to animate a stack data structure, "OopaStack" class from the animation library can be used. Whenever an operation is done on the user data structure, the corresponding operation should be done on the animation object. To do this, a message has to be sent to the animation object inside the function definition of the user data structure. For example, if the user is trying to animate a stack data structure, the user class may look like the code in figure 3.

```
Class UserStack {  
    ...  
    ...  
    ...  
    ...  
    OopaStack mystack;  
    public :  
    ....  
    ....  
    ....  
    void push(...) {  
    ....  
    ....  
    ....  
    mystack.OopaPush();  
    }  
}
```

Figure 3. Example of a User Class that Animates a Stack Data Structure.

In the above example, the user first declares an instance of OopaStack. The user sends the push message to mystack inside the push operation of the UserStack. Similarly other operations of UserStack can be animated.

Once the program is written with animation objects, the program has to be precompiled. This can be done by the following command :

```
PRECOMPILE USERPROG.C OUTPUT.C
```

Then compile output.c using BorlandC++ 2.0 or later. Then the output.exe file under Windows is run to see the animation. This opens a window with scrollbars and a menu at the top of the window.

Graphical User Interface

The graphical user interface mainly consists of a window with horizontal and vertical scrollbars, and a menu.

Scrollbars

The user can scroll to the right by clicking on the left arrow on the scrollbar and to the left by clicking on the right arrow. Similarly the user can scroll vertically by clicking on the up and down arrows.

Mainmenu

The menu can be accessed by pressing the ALT key and the underscore letter or by pointing and clicking with the mouse. The animation starts when the user selects RUN under the FILE menu. When the user selects SPEED

under the options menu, a dialog box with a scrollbar control comes up. The user can increase or decrease the speed of animation by clicking the left or right arrow of the scrollbar control. Keyboard arrows can also be used. Speed is initially set to medium speed. If the user changes the speed but decides not to save it, the user can click the CANCEL button. The ABOUT item under SPECIAL will open up a dialog box which gives a brief information about the package and its creator. It also has the OODSA icon in it.

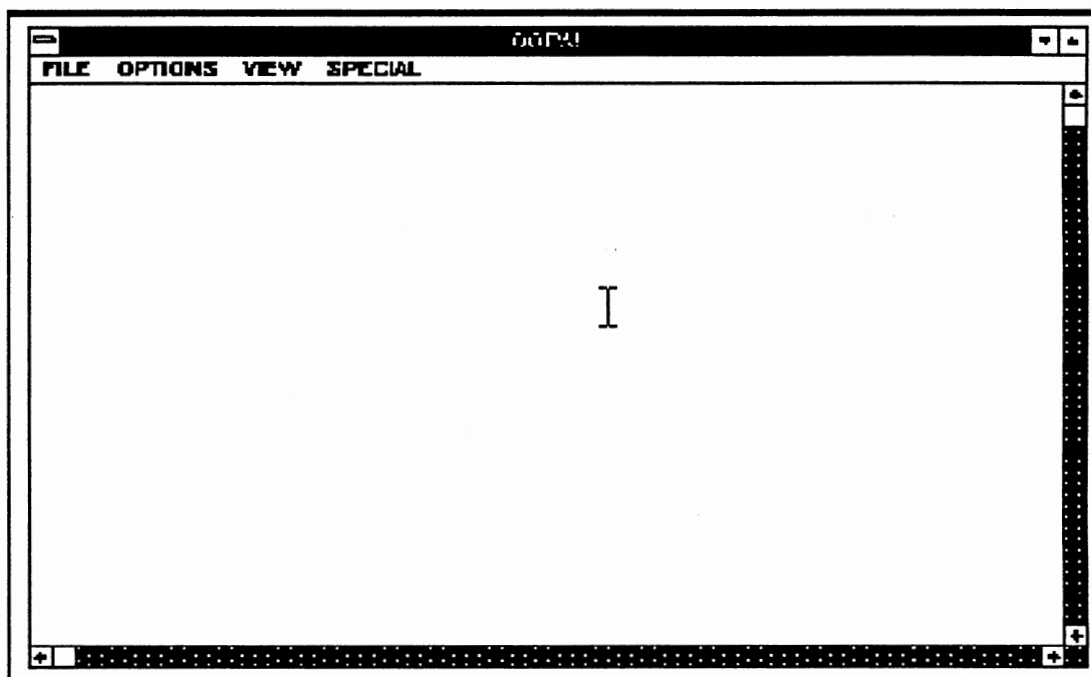


Figure 4. The Main Window of the GUI

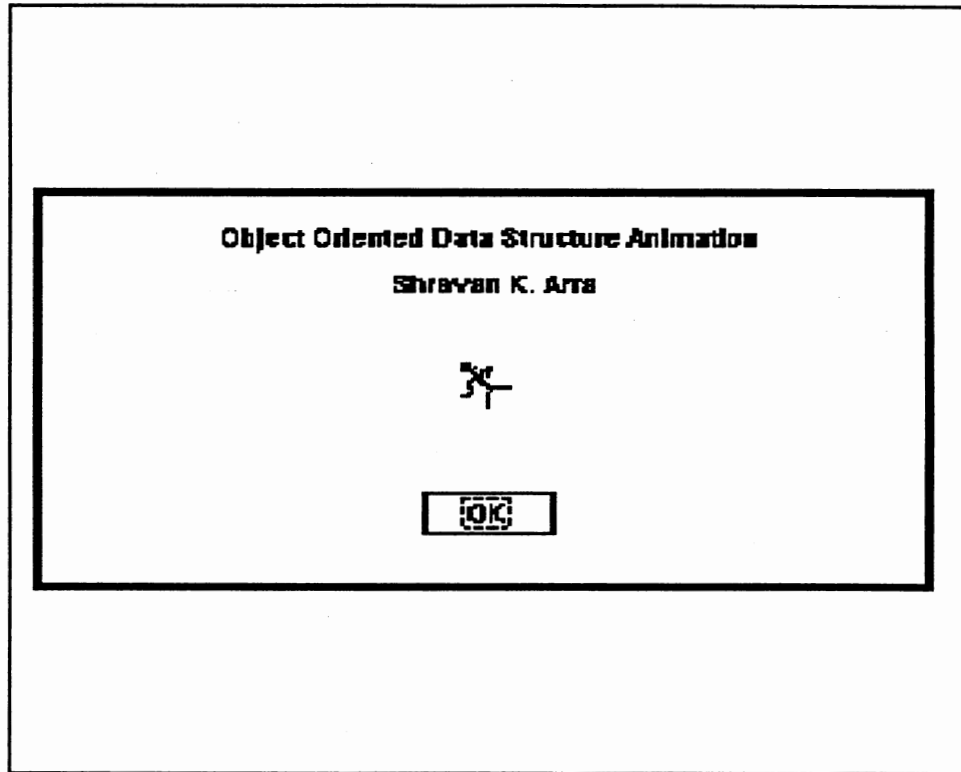


Figure 5. The About Dialog Box

Animation Library

The Animation Library is divided into two groups :

- 1) Basic object classes
- 2) Data structure classes

The Basic object classes group contains the basic objects needed for animation. This prototype provides a graphic object class and an arrow class in this group. Most data structures can be drawn by using these classes. The data structure classes group contain the data structure animation classes that use the basic object classes.

The graphic object class draws a bitmap on the screen moves it horizontally

or vertically, changes its width or height, etc. The package provides two shapes(rectangle, circle) and either of these shapes can be used. The package currently provides only straight paths.

OopaRect Class

The OopaRect Class provides the bitmaps needed for animation. The following table gives the attributes of a bitmap and operations that can be done:

OopaArrow Class

The OopaArrow Class provides the arrows needed to represent pointers. The following figure gives the attributes of an arrow and operations that can be done on them.


```
Class OopaRect {  
    int topx;  
    int topy;  
    int width;  
    int height;  
    Color color;  
    char text[25];  
    char shape[20];  
    public :  
        DrawRect(.....);  
        MoveRectHoriz(...);  
        MoveRectVert(...);  
        ChangeRectHeight(...);  
        ChangeRectWidth(.....);  
        ChangeRectColor(...);  
}
```

Figure 6. The OopaRect Class.

```
class OopaArrow {  
    int topx;  
    int topy;  
    int width;  
    int height;  
    Color color;  
    int direction;  
    public :  
        DrawArrow(.....);  
        MoveArrowVert(distance);  
        MoveArrowHoriz(distance);  
        ChangeArrowColor(color);  
}
```

Figure 7. The OopaArrow Class

APPENDIX B

WINDOWS PROGRAMMING

Introduction

History

Microsoft Corporation released Windows 1.0 in 1985 and Windows 2.0 in 1987 after several updates to Windows 1.0. The most significant improvement in Windows 2.0 was the introduction of overlapped windows in place of tiled windows. Windows 3.0 was introduced on May 22, 1990. The enhancement in Windows 3.0 is the support of protected mode operation of Intel's 80x86 microprocessors. The latest release of Windows is Windows 3.1.

Why Windows

Users generally spend a lot of time learning a new software package. But with Windows this amount of time is reduced because all Windows packages have a consistent user interface. Once a user learns how to use one Windows application, learning a second one is very easy. Programs written for Windows do not directly access the hardware of devices, such as video display and a printer. Windows has a graphic library that allows easy access to the hardware. The functions in this graphic library are written in such a way that they will function with

any device. Hence Windows programs are portable. Windows is a multitasking environment where more than one program can be running at a given point of time.

Message-Driven Architecture

At any time, the user using a Windows application can do many things, such as, resize a window, minimize a window, close a window, etc. For each action the user performs, the windows application must act accordingly. This is efficiently done in Windows using a message driven architecture. With this, whenever a user issues a command, a message is sent to the Windows application. Windows has queues that receive the messages. It has a system queue (also called Hardware Event Queue) that receives messages from keyboard, mouse and timer. It also has a separate application queue for each running application that receives other types of messages. Windows appropriately copies messages from system queue to the application queue. Each application will retrieve messages from its application queue by using GetMessage function. The messages in the application queue of the active application are processed first. If there are no more messages in the active application queue, messages from other application queues are processed. It is through this messaging system that Windows achieves its multitasking capabilities. Apart from queued messages there are non-queued messages that are directly sent to a window procedure. The window procedure is called as if it was a subroutine that was a part of Windows. In a traditional

multitasking system, each application is allotted a time period. If the application takes more time than the allotted period, it is preempted and another application is processed. But in Windows, once Windows starts processing a message it cannot go to the next application until the current one is finished. Hence Windows is a non-preemptive multitasking environment.

Memory Management

Memory management is essential in a multitasking environment. When more than one application is running, an application may erroneously read or write data that belongs to another application. So a memory management scheme that prevents such errors is needed.

Movable Memory

When a block of memory is allocated, a program receives a handle that identifies the memory instead of a fixed address. Such a block of memory can be moved when needed to decrease fragmentation. But this is transparent to the programmer because he can still access the same memory using the same handle.

Discardable Memory

If more memory is needed, the Windows memory manager can do more than move blocks of memory to reduce memory fragmentation. It can discard blocks of memory that can be reloaded. An example of discardable memory is

code. Since code is not modified during the execution of a program it can be reloaded when needed.

Fixed Memory

There are situations that require memory be not moved or discarded. For example, an interrupt handler will require a fixed location in memory since it must always be ready to process an interrupt. The use of fixed memory should be limited to special cases like device drivers.

When multiple instances of the same program are running, Windows uses same code segments and same resources for all instances but different data segments for each instance. Code segments and resources are demand loaded and discardable in Windows.

Windows runs in three different modes. They are : 1) Real mode 2) Standard mode 3) Enhanced mode.

Real Mode

In this mode the logical address is equal to the real physical address. Only one megabyte address space is available. This runs on all 80xx processors. Prior to Windows 3.0 all versions of Windows ran only in real mode. Real mode is fast, but the problem with real mode addressing is that it makes it very difficult for an operating system to manage memory.

Standard Mode

Standard mode gives Windows the benefits of protected mode on the 80286 and the 80386 processors. In this mode, Windows programs get a physical address space of up to 16 megabytes instead of the 1 Megabyte limit of Real Mode.

Enhanced Mode

In this mode Windows gets all the benefits of standard mode and a larger address space. In this mode, the address space can grow to a size that is up to four times the available physical memory by using virtual memory techniques.

Both standard mode and enhanced mode run under the protected mode of the 80x86 processor. In protected mode certain rules are enforced when memory is being accessed. If a program tries to access a memory location, intentionally or accidentally, that does not belong to it, a CPU interrupt is generated. In Windows, this results in the abnormal termination of the offending program and a UAE(Unrecoverable Application Error message) being displayed. This helps in maintaining order in a multitasking environment.

Graphics Device Interface

Graphics in Windows are handled primarily by functions exported from the GDI.EXE module. GDI.EXE module calls various driver files in turn. GDI handles graphic output for the display screen as well as hardcopy devices such as printers and plotters. In addition to physical devices like video screens and

printers, GDI supports logical or pseudo-devices. Logical devices store a picture in RAM or on disk.

Device Context is the collection of current "attributes" that determine how the GDI functions work on the device. The attributes are for example, foreground color, background color, font etc. To be able to draw on a particular device, a program must first obtain a handle to its Device Context of that device, shortly called DC. When something is drawn on the device, the current attributes of its DC are used. For example when you display text on a screen, you don't have to specify the font and the foreground color. Windows uses the font and the color in the device context of that screen. There are several functions to get and change the attributes of a device context.

Memory Device Context

A memory device context is a device context that has a display surface that exists only in memory. Almost everything that can be done with a device context can be done with a memory device context. When a memory device context is created it has a display surface that contains exactly one monochrome pixel. To make the display surface of the memory device context larger, a bitmap is selected into the memory DC. Now the memory DC has the same area as the bitmap. If the bitmap had a picture on it, then that picture is now part of the memory DC's display surface. Any changes that are made to the bitmap are reflected in this display surface. A memory device context can be used to draw an object in memory and do manipulations to it before drawing

it on a real device.

Bitmaps

A bitmap is a digital representation of a picture. Each pixel in the image corresponds to one or more bits in the bitmap. Monochrome bitmaps require only one bit per pixel; color bitmaps require additional bits to store the color of the pixel. A bitmap can be constructed using Microsoft's SDKPAINT, Borland's WRT or by setting the bits of the bitmap in a program. In this thesis we use Borland's WRT to draw bitmaps. Bitmaps can be drawn, compressed or stretched on the display device. The function BitBlt does this. Blt(pronounced as blit) means block transfer. This function moves a block of bits from one device context to another. To draw on the screen, select the DC of the window as the destination device context. StretchBlt function can be used to draw a stressed or compressed bitmap. These two functions take the coordinates in terms of logical units.

VITA

Shravan K. Arra

Candidate for the Degree of

Master Of Science

Thesis: OBJECT-ORIENTED DATA STRUCTURE ANIMATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Hyderabad, India, Feb 17, 1968, the son of Srihari Arra and Vimala Devi.

Education: Received Bachelor of Technology degree in Electrical Engineering from Jawaharlal Nehru Technological University, India, in 1989. Completed degree requirements for Master of Science in Computer Science in July 1992.

Professional Experience: Lab Assistant, Office of Business & Economic Research, Oklahoma State University, May 1990 to May 1992.