

AN ABSTRACT OF THE THESIS OF

Sudheendra S. Gulur for the degree of Master of Science in Mechanical Engineering presented on October 7, 1994. Title: SketchPad for Windows: An Intelligent and Interactive Sketching Software.

Redacted for Privacy

Abstract approved:



David. G. Ullman

The sketching software developed in this thesis, is aimed to serve as an intelligent design tool for the conceptual design stage of the mechanical design process.

This sketching software, *SketchPad for Windows*, closely mimics the traditional paper-and-pencil sketching environment by allowing the user to sketch freely on the computer screen using a mouse. The recognition algorithm built into the application replaces the sketch stroke with the exact CAD entity. Currently, the recognition of two-dimensional design primitives such as lines, circles and arcs has been addressed.

Since manufacturing requires that the design concepts be detailed, sketches need to be refined as detailed drawings. This process of carrying design data from the conceptual design stage into the detail designing stage is achieved with the help of a convertor that converts the sketch data into DesignView (a variational CAD software). Currently, only geometrical information is transferred from the sketching software into DesignView.

The transparent graphical user interface built into this sketching system challenges the hierarchial and regimental user interface built into current CAD software.

SketchPad for Windows
An Intelligent and Interactive Sketching Software
by
Sudheendra S. Gulur

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed October 7, 1994
Commencement June, 1995

Master of Science thesis of Sudheendra S. Gulur presented on October 7, 1994

APPROVED:

Redacted for Privacy

Major Professor, representing Mechanical Engineering

Redacted for Privacy

Chair of Department of Mechanical Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Sudheendra S. Gulur

ACKNOWLEDGEMENTS

This thesis could not have been completed without the help of a number of people. I would like to thank , most of all, my parents for their love and support. I would also like to thank my relatives (Sathya Rao and family, Hiranyappa and family, Jayyi, Harisarvotham and family, Vasant Lambu and family, Nagesh Rao and family, Shakuntala Bai, Krishna Rao and family, Rama Rao and family, Venkatesh and family) for their encouragement. I will be failing in my duties if I do not thank the Bhaskars for their hospitality and encouragement.

I am grateful to Dr. David G. Ullman for guiding me and for having provided the necessary infrastructure for the successful completion of this thesis.

Special thanks go to Ashutosh Kale and Nageswara Rao Cheekala for their constant support, encouragement, and assistance in debugging the software and to Arun and Chitra for their guidance on data structures and object-oriented programming.

Last but not the least, I would like to thank all those who made my stay in Corvallis enjoyable.

TABLE OF CONTENTS

	<u>Page</u>
1. INTRODUCTION	1
1.1 Current computer solutions for the mechanical design process . . .	1
1.2 SketchPad for Windows - The concept	2
1.3 SketchPad for Windows - The implementation	5
1.4 SketchPad for Windows - An illustration	5
1.5 Organization	6
2. IMPORTANCE OF SKETCHING	8
2.1 Introduction	8
2.2 Design sketches	9
2.3 Refining design sketches	10
2.4 CAD for conceptual design?	11
2.5 Research in the area of sketching	12
2.6 Research on sketching software in this thesis	14
3. PROGRAMMING SKETCHPAD FOR WINDOWS	15
3.1 Object-oriented programming	15
3.2 Windows programming concept	26
3.3 Object Windows Library (OWL 2.0)	33
3.4 Resources and graphical user interface development	48
4. SKETCHPAD FOR WINDOWS	56
4.1 Introduction	56
4.2 Motivation for implementing SketchPad for Windows	57
4.3 Sketchpad for Windows	60
4.4 Feature implementation	76
5. CONCLUSIONS	93
5.1 Summary	93

5.2	Limitations of SketchPad for Windows	98
5.3	Recommendations for future research	99
6.	BIBLIOGRAPHY	100

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1.1 Sketching interface	3
1.2 Role of a sketching system in the mechanical design process	5
1.3 Linear trail	6
1.2 Line recognition	6
2.1 Different forms of pictorial representation	9
2.2 Different sketching stages	11
3.1 Class hierarchy	16
3.2 Real world definition of a four wheeled automobile	18
3.3 Multiple inheritance	24
3.4 Input messages	30
3.5 Windows operation	31
3.6 Basic OWL window	35
3.7 Improved OWL window	37
3.8 Menu implementation	40
3.9 Open/Save Dialog Box	44
3.10 Speedbars in applications	47
3.11 Windows resources	49
3.12 Visual representation of resources	50
3.13 Resource binding	51
3.14 Menu editor	51
3.15 Dialog Box editor	52

3.16 Bitmap editor	53
3.17 Stringtable editor	55
4.1 Design Capture System	57
4.2 Inking process	58
4.3 Drawing information	59
4.4 SketchPad for Windows icon	61
4.5 SketchPad for Windows	61
4.6 Design information in the mind	62
4.7 User messages	63
4.8 Linear sketch trail	64
4.9 Line recognition	65
4.10 Base plate - Stage1	65
4.11 Bolt hole sketching	66
4.12 Circle recognition	67
4.13 Entity selection	67
4.14 Entity modified	68
4.15 Arc trail	69
4.16 New design	70
4.17 Importing the sketch into DesignView	72
4.18 Dimensioning the drawing	73
4.19 Dimension driven geometry	73
4.20 Capturing sketch information	74
4.21 Design Capture in Solution Library	75

4.22 Window objects in SketchPad for Windows	77
4.23 Window classes in SketchPad for Windows	77
4.24 Link list	84
4.25 Class hierarchy	87
4.26 Sketch recognition algorithm	91
5.1 Transparent graphical user interface	93
5.2 Linear trail	94
5.3 Line recognition	94
5.4 Design Capture System	96

LIST OF TABLES

<u>Table</u>	<u>Page</u>
3.1 Window messages	32
3.2 Response functions and Window messages	38
3.3 Command messages	43
3.4 Cursors and their use	54
4.1 Recognition parameters	90

SketchPad for Windows

AN INTELLIGENT AND INTERACTIVE SKETCHING SOFTWARE

1. Introduction

This master's thesis studies the development of computer tools as design aids for the conceptual design stage of the mechanical design process. It presents to the user an intelligent and interactive sketching tool called *SketchPad for Windows* that closely mimics the pencil and paper sketching environment. Section 1.2 justifies the inability of CAD software to support sketching tools for the conceptual design stage of the mechanical design process (see section 1.1) while introducing the concept of offering a transparent user-interface for sketching in *SketchPad for Windows*. An overview of the development environment is then explained in section 1.3.

1.1 Current computer solutions for the mechanical design process

Computer Aided Design and Computer Aided Manufacturing (CAD/CAM) plays a major role in the area of product design and manufacturing. Today, there are CAD/CAM software that allow the user to design and visualize products, perform analysis (stress, kinematic, dynamic *etc.*), and prototype rapidly. Research in the area of CAD/CAM has resulted in excellent software for the detail designing and manufacturing process.

There is a marked difference in the way the designers design in the pen and pencil environment than when they design on a CAD software [Gulur 92]. This is because, the additional cognitive load to implement current CAD systems is detrimental to the design process, *i.e.*, the icon and menu selection add an unnecessary step in the creative thinking process [Ullman 90].

Learning to use CAD software is a slow process requiring the designer to condition his/her mind to design in the regimental CAD environment rather than the normal

design process using paper and pencil. This slow learning process is mainly due to the fact the current CAD systems drive the designer's thought rather than the designer driving the CAD software [Waldron *et.al.* 88]. In other words, the designs that are accepted on CAD systems require that they be in detailed form with dimensions. True conceptual design (see chapter 2 on sketching), does not start with ideas containing detail dimensioning. Instead, sketches are used to support idea development during the conceptual design stage [Fang 88, Ullman 90, Hwang 91, Luzzader 75]. Therefore, most design engineers using CAD systems for conceptual design regress to sketching on a piece of paper before feeding data to CAD systems. Moreover, the entity (line, circle, arc *etc.*) icons and menus used in CAD systems require that the designer think in terms of these entities rather than in terms of sketches.

Since most designs start as sketches, it is necessary for CAD software to allow the designer to express his/her design thoughts as free-form sketches. In other words, CAD software do not address the conceptual design stage. What is then required is a computer aided sketching tool that closely mimics the normal sketching process and allows the designer to express his/her thoughts in the form of sketches.

1.2 SketchPad for Windows-The concept

SketchPad for Windows has been developed to aid the conceptual design stage of the mechanical design process.

One of the distinguishing features of *SketchPad for Windows*, is it's user interface that allows the designer to express his/her thoughts in the traditional manner. Since the traditional paper-and-pencil sketching environment utilizes a minimum of drawing tools, simulation of a paper-and-pencil environment in a sketching software requires a graphical user interface with *transparent* sketching tools. The term *transparent* is

used as an adjective to describe a non-regimental, non-hierarchical user interface. The objective of implementing such a graphical user interface is to eliminate the standard drawing entity icons (line, circle, and arc) and instead, implement one interface that can handle drawing of all basic sketching primitives. As can be seen in the Figure [1.1], *SketchPad for Windows* supports only one interface for sketching.

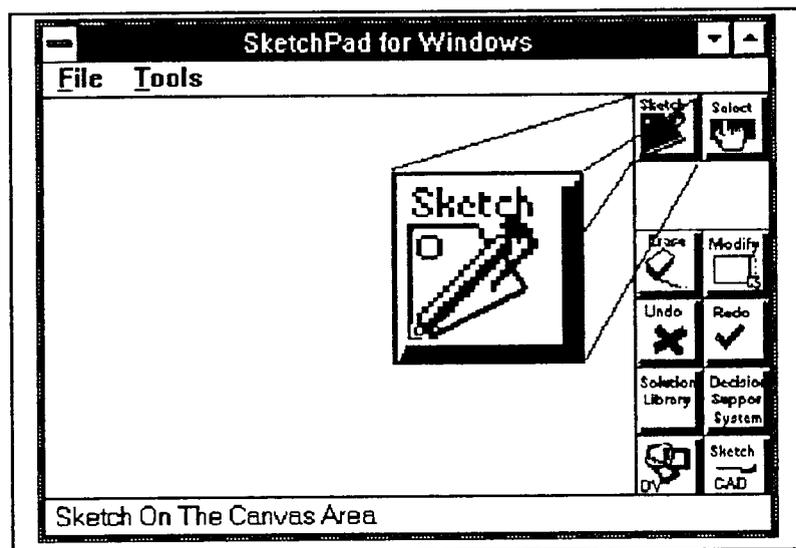


Figure 1.1: Sketching interface

The designer strokes with the mouse on the screen, as he does in the a pencil-and-paper sketching environment, and *SketchPad for Windows* recognizes the stroke and replaces it as a CAD entity. For example, sketching in a linear fashion causes *SketchPad for Windows* to recognize the stroke as a line.

SketchPad for Windows also allows the designer to capture bitmapped images of the sketches made on the screen. These bitmapped images can be used to:

1) Define or capture the design functionality using the *Solution Library* [Wood 95]. The function-driven *Solution Library* accepts bitmapped images and associates them with functions so that the function can be used as a search word to retrieve design solutions.

2) Compare alternative sketches using the *Decision Support System* [Herling *et al.* 94]. *Decision Support System*, a decision analysis database tool, is based on the IBIS (Issue Based Information System) [Rittel 73] and OREO (Object Relation Object) [Ullman 93] theory. It allows the design engineer to solve complex design issues. The issue is an identified problem to be resolved by arguing the pros and cons of proposed alternatives. Sketches by different designers can be stored as different issues to aid comparison of designs.

No design has commercial value unless the object designed can be manufactured. Manufacturing requires a detailed design. As mentioned earlier, CAD software are ideally suited for representing detail drawings. Consequently, sketches need to be refined into detail drawings. *SketchPad for Windows* allows the designer to convert the sketches into CAD data. Currently the conversion utility allows the designer to convert the sketch data into DesignView CAD data so that he/she can perform parametric design calculations within DesignView.

SketchPad for Windows, *Solution Library*, and *Decision Support System* assist in the conceptual design stage of the mechanical design process, while DesignView assists in the detail designing stage. The integration of *SketchPad for Windows*, *Solution Library*, *Decision Support System* and DesignView is as shown in the Figure [1.2].

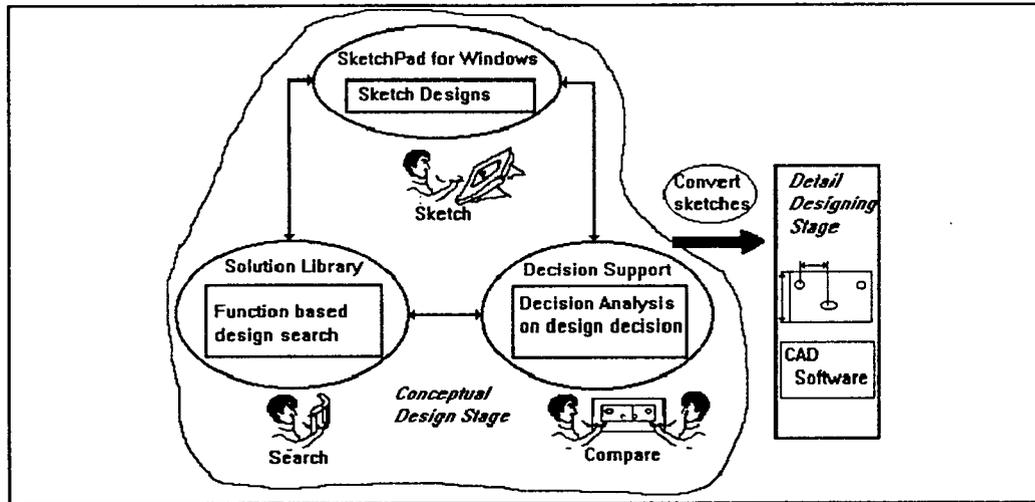


Figure 1.2: Role of a sketching system in the mechanical design process

1.3 SketchPad for Windows-The implementation

The *Solution Library*, *Decision Support* system, and *DesignView* CAD software operate on a PC (Personal Computer) and run under the Microsoft Windows environment. Therefore, SketchPad is also developed for the Microsoft Windows environment.

Microsoft Windows applications can be developed with the commercially available development kits from Borland International and Microsoft Corporation. Borland's *Object Windows Library* (OWL 2.0) was used to develop the application. Implemented in C++, OWL employs the principles of object-oriented programming.

1.4 SketchPad for Windows-An illustration

The user sketches as he would sketch on a piece of paper, and the sketch recognition algorithm recognizes what the user intended. The impressive feature of the

recognition algorithm is the fact that the freehand sketches are corrected as they are drawn. For example, if the user stroked in a linear fashion (Figure 1.3) keeping the left mouse button down, and released the left mouse button upon completion of the stroke, *SketchPad for Windows* recognizes the trail and replaces the sketch with a line (Figure 1.4).

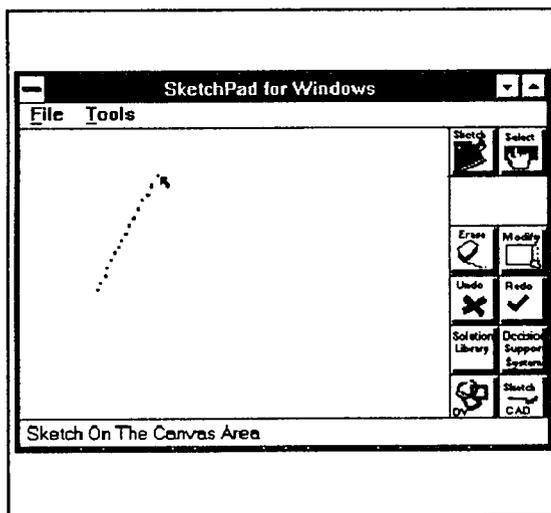


Figure 1.3: Linear trail

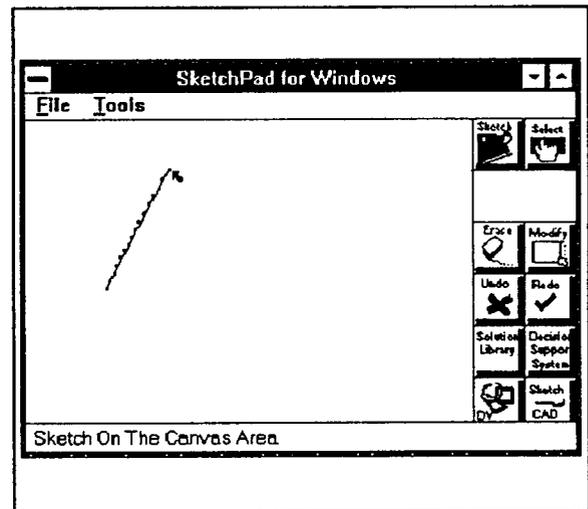


Figure 1.4: Line recognition

Similarly, arcs and circles are recognized from freehand sketches that the user makes. As can be seen from the example shown above, *SketchPad for Windows* allows the user to represent design ideas without having to think in terms of the respective entity icons.

1.5 Organization

Since this thesis aims to present to the user a powerful, and intelligent sketching

software, chapter 2 describes the importance of sketching in the design process. This chapter also describes briefly the current sketching software developed at research institutions.

The concepts of object-oriented programming necessary to understand and implement *SketchPad for Windows* using Borland's object-oriented Microsoft Windows application development kit *OWL* are discussed in section 3.1. As will be explained in chapter 3, this new programming methodology allows the user to think and program in terms of objects that he sees around him in this world. Microsoft Windows application development is based on the event based programming methodology and is explained in the section 3.2. Microsoft Windows application development with *OWL* is described in section 3.3. Microsoft Windows environment offers a standard graphical user interface (GUI) to all applications and hence GUI development will be explained in the section 3.4 on Resources and Graphical User Interface development.

A demonstration of the use of *SketchPad for Windows* in the conceptual design stage to capture design sketches is explained in the section 4.3 while section 4.4 details the application and algorithm implementation.

The modular nature of the code for *SketchPad for Windows* makes room for changing the inner details of the sketch recognition modules when new algorithms and approaches are discovered. The chapter on suggestions and recommendations at the end of the thesis discusses this in detail.

2. Importance of sketching

Since this thesis provides the user with an intelligent and powerful sketching software, this chapter describes the importance of sketching in the design process. Starting from a discussion on the different forms of pictorial representation, this chapter blends into a discussion on the importance, and different stages in sketching designs. Questioning the use of CAD in the conceptual design stage (section 2.4) of the mechanical design process, research engineers have strived to develop software tools that aim to mimic the normal sketching process (section 2.5). Deriving inspiration from some of the earlier works on sketching software, section 2.6 discusses the specifications for implementing *SketchPad for Windows*.

2.1 Introduction

Pictorial representations have, from early times, been the means of conveying the ideas of one person to another or from one group to another. Primitive man made sketches on cave walls which today convey information about their lives. Nothing can replace a picture when people communicate from different bases of knowledge and experience [DeJong 83]. Luzzader [Luzzader 75] describes the graphic powers of the various forms of pictorial representation [Figure 2.1] and writes....

"The use of sign language representation, shown in (a), is rather easy to learn and may be quickly executed but interpretation is restricted to persons who understand the particular language in which it is presented. The multi-view representation, shown in (b), may be understood universally by persons who have been trained in its use. However the given views will prove to be almost meaningless to many who have not had the advantage of needed training. What then is the one form of representation that can be understood and used by all? It is the pictorial form shown in (c)."

Being able to graphically represent (pictorially or multi-view) one's thought helps in organizing one's creativity. Sketching helps to stimulate one's imagination as well as

convey information. Many a times the designer/engineer finds himself/herself in a position where-in he/she finds it easier to express thoughts through sketches rather than through words or mathematically.

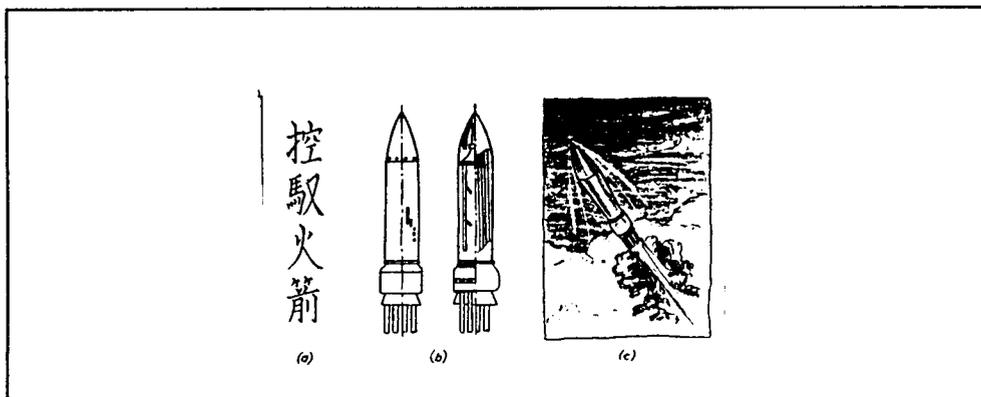


Figure 2.1: Different forms of pictorial representation [Luzzader 75]

2.2 Design sketches

Sketching acts as an extension of the designer's short term memory, helping to extend ideas that might be otherwise forgotten [Ullman 90]. Dejong [Dejong 77] aptly explains this graphical form of expressing creativity..."In most engineering offices, when a new idea or fine point has to be discussed, someone will laughingly request some talking paper to sketch a graph or section view, or just to elaborate on an equation."

Sketches serve as an external record of the visual image in the mind of the designer and are a principal medium of external thinking [Herbert 87]. Bally [Bally 87] emphasized the importance of sketches in the design process by stating:

"If a designer has several ideas in mind, he typically can't evaluate the relationship among them until they are made visible. A drawn external representation is a very important aid to the designer in making spatial interference..... In design, the designer's current sketches are used as a source of ideas and interference which shape later design decisions. Because the partially completed product is continually changing, the task environment is continually changing. These changes in environment stimulate new ideas and inferences... We have observed that designers do not know many details about what they are going to sketch until part of the sketch is made."

Sketches, often serve to clarify an original idea and at the same time suggest additional versions, adaptations, and improvements. As ideas are generated, the designer starts sketching to organize his thoughts and clearly visualize the problems that arise. The preliminary ideas (stored in pictorial form) along with the orthographic sketches are recorded at every stage. Very often pictorial sketches of some detail of construction will prove to be more intelligible and will convey the idea much better than an orthographic sketch [Luzadder 75].

2.3 Refining design sketches

Most designs start with the conceptualization stage and is represented by a sketch. As the sketch gets more and more refined [Figures 2.2a,b and c], the designer checks the functionality of the design through layout drawings. For example, a kinematic design layout allows the user to verify the functionality of the designed mechanism. While most layout designs end up as detail designs, some need further clarification (manufacturing dimensions) before they can be called detail drawings. These detailed drawings could then be used for manufacturing the product.

Since the sketches go through the refinement stages referred to in the previous paragraph, it is worth analyzing the use of CAD in the sketch capture and refinement stages of the conceptual design stage of the mechanical design process.

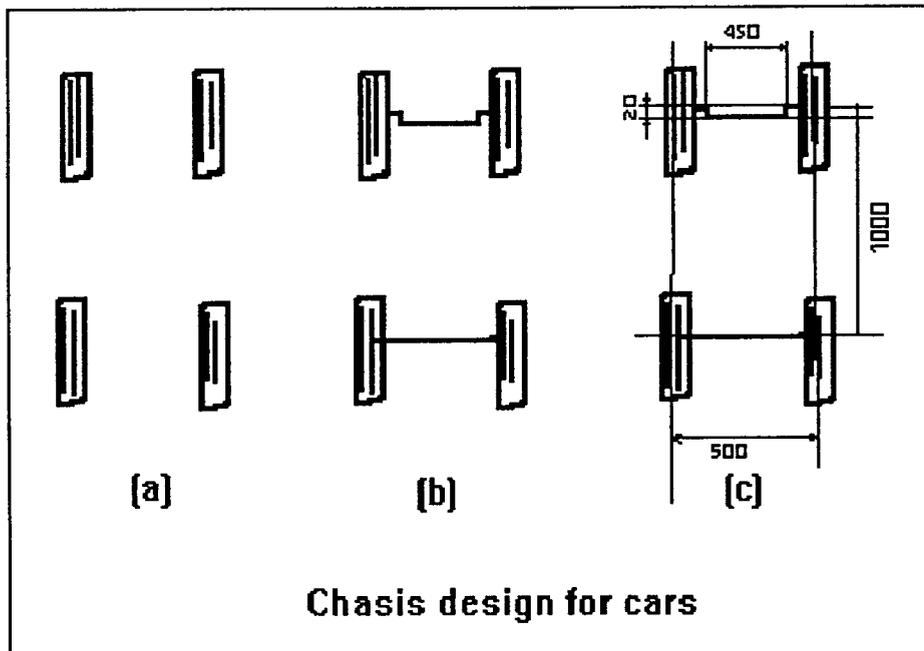


Figure 2.2: Different sketching stages

2.4 CAD for conceptual design?

Design sketches differ from ordinary sketches in that they are not meant to be artistic in nature. In other words, design sketches are meant to be simple sketches of a design idea and hence lack artistic refinement. The process of sketching, according to Ryan [Ryan 90], is to be quickly done without the use of instruments or guides as it is a symbolic representation of the real world object and is not intended to be perfect and is supposed to look freehand. Luzadder [Luzadder 75] emphasizes that while sketching, the mind should be centered on the idea and not on the technique of sketching. Although the sketch is freehand and lacks refinement by mechanical instruments, it is based on the same principles of projection and conventional practices that apply to multi-view, pictorial, and the other divisions of mechanical drawing.

A study of the importance of drawing in the design processes [Ullman 90] showed that:

- * 72% of the marks made on the paper constituted sketching and drafting, while dimensioning comprised 14%, textual information 9% and 5% calculations.
- * During the conceptualization stage most of the graphics is 2D (59%), in the layout stage it is a mixture of 2D and orthographic, while in the detail phase it is comprised mostly (78%) of orthographic drawings.

As can be seen from the results of the above mentioned study, graphics in the conceptual design stage is predominantly 2D. Moreover, 67% of the drawings constituted sketches [Ullman 90]. One of the striking features of conceptual design is the fact that it requires no detail dimensions. However, present day CAD software and feature based modelers offering excellent graphics routines claim that they are useful for conceptual design. These systems require that the design be detailed with dimensions. Because of the mismatch in the input parameters that these CAD software and feature based modelers take and the data that the designer has in his mind as free-hand sketches with no dimensions, these systems are not ideal for the design conceptualization stage.

2.5 Research in the area of sketching

Convinced about the utility of sketching in the conceptual design stage, researchers are trying to develop sketching environments on computers, that allow capturing of designer's thoughts.

Some researchers suggest capturing of design sketches by:

- 1) digitizing the sketch on the paper and converting it into a CAD model [Suffel and Blount 89]. However, digitizing requires re-entering of data into the computer and is hence not an ideal solution.
- 2) Hough transforming the sketch on the paper through image processing [Kasturi

*et.al.*90]. Hough Transformation, an image processing routine, is performed on detailed sketches to recognize sketches. This application does not recognize free-hand sketches for image processing and is therefore not an ideal solution too.

- 3) Extracting featured points, lines and curves from a hand-written drawing through image processing [Iwata *et.al.* 87]. However, this application requires that the user scan the hand-written drawing with a "drum scan densitometer" before being able to extract the feature points, lines, and curves from the drawing and is therefore not ideal.

One of the recent applications, Easel [Jenkins *et.al.*, 92], attempts to mimic the traditional sketching environment. Written in C programming language, Easel runs on X-Window. It accepts the mouse trajectory as input and tidies the sketch by replacing it with an appropriate geometric entity. The 2D sketch recognition by Fang [Fang 88] also attempts to mimic the sketching process.

Easel allowed the user to sketch on the screen with the mouse and the system would replace the sketch-stroke with an appropriate CAD entity. One of the striking features of Easel was its ability to copy portions of the sketch to the clipboard. Although Easel supported geometrical reasoning features, it did not support entity modification features that were needed for sketch refinement.

Fang's [Fang 88] sketch recognition system was developed for the UNIX based HP workstation. This sketch recognition system allowed the user to sketch with a locating pen on the tablet. The strokes were then represented as CAD entities. Although Fang's work was later incorporated into Hwang's [Hwang 91] research, these sketching systems could not open or save the sketches, nor did it allow entity modification other than deletion. Moreover, since these sketching systems worked as stand-alone applications and required specific device drivers for operation, their use as a general purpose sketching tool was limited.

2.6 Research on sketching software in this thesis

The primary objective of the implementing *SketchPad for Windows* was to develop a sketching system similar to Easel, based on recommendations in Fang's work, which allowed the user to modify entities, perform file operations and was not device-driver dependent. Microsoft Windows and X-Windows are graphical user interface environments that support a host of device-drivers.

Since computer tools for design are now offered more as desktop PC-based Microsoft Windows solutions rather than X-Window based solutions, it was decided to develop *SketchPad for Windows* for the Microsoft Windows environment. Using *Object Windows Library* (OWL), the application development kit from Borland, *SketchPad for Windows* was developed in C++, the superset of the C programming language.

3. Programming SketchPad for Windows

As mentioned in section 2.5, *SketchPad for Windows* has been written in Borland C++ using the Microsoft Windows application development kit, *Object Windows Library* (OWL 2.0). OWL is also written in Borland C++ and is based on the object-oriented programming methodology. The event-based Microsoft Windows programming methodology is discussed in section 3.2. Section 3.3 on *Object Windows Library* discusses the implementation details of the "Window objects" in this application development kit. Finally, since Microsoft Windows offers a standard user interface to all its applications, section 3.4 discusses the standard resources and graphical user interface development for Microsoft Windows applications.

3.1 Object-oriented programming

An object is defined in the Webster's dictionary as an inanimate thing that has a fixed shape or form and which can be touched and seen. Any object in this real world can be classified to be belonging to a particular group. For example, the Ford Taurus and Ford Escort belong to the category of new generation Ford cars while the Toyota Camry and Corolla belong to the category of new generation Toyota cars. While the above mentioned objects belong to the category of mechanical objects, an object in the computer environment may consist of:

- * Windows
- * Menus
- * Dialog Boxes
- * Graphic objects (lines, circles, rectangles, arcs etc)
- * The mouse and keyboard

All objects in this real world have properties and behaviors. For example, a car is made of steel (property) and can be used to move from one place to another

(behavior). In a similar manner the dialog box in a computer environment has a defined size, color (property) and can be moved, and closed (behavior). Thus object-oriented programming requires the developer to think in terms of how a program will be divided into objects, properties, and behaviors. Goldstein and Alger [Goldstein and Alger 92] describe in simple words the concept of object-oriented design as..."Object Oriented Design (OOD) expresses software designs in terms of objects and their properties, with the hope that non programmers can understand and comment on the design ."

The basic idea behind the object oriented programming approach is the way the data and functions operating on the data are combined into one unit called an **Object**. An object's functions, called member functions in C++, typically provide the only way to access it's data, thus hiding the data from misuse. Objects are grouped into **classes** based on their shared properties. For example, cars and trucks can be grouped into a class of four wheeled automobiles [Figure 3.1].

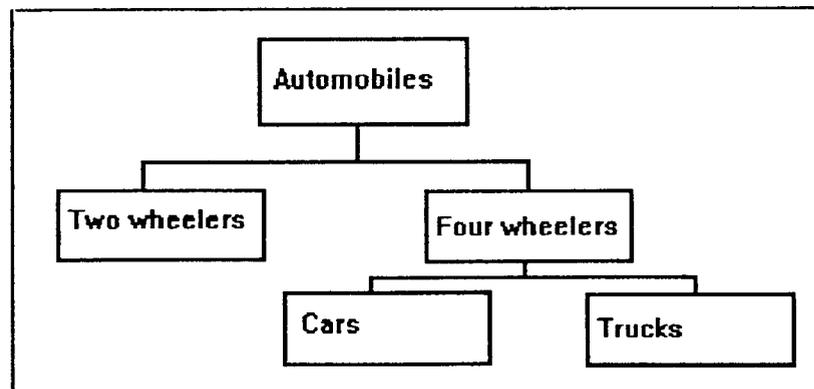


Figure 3.1: Class hierarchy

As can be seen from Figure [3.1], the two categories of automobiles (two wheelers and four wheelers) can be grouped into a superclass (automobiles) based on shared properties.

Objects communicate with one another. An automobile intending to make a turn indicates or communicates to other automobiles by flashing the appropriate indicator light. The other automobile then takes the necessary action such as slowing down or yielding. In a similar manner, in the computer environment, clicking on a menu button to display an object sends a message to the graphic object required to be displayed. Calling an object's member function is referred to as sending a message to the object. The **method**, or member function, invoked by the receiver object in response to an incoming message is decided by the class to which the object belongs. Budd [Budd 91] explains the process of transmission of messages as:

Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The message encodes the request for an action, and is accompanied by any additional (arguments) needed to carry out the request. The receiver is the agent to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

Classes can be organized into a hierarchy **inheritance** structure with each derived class inheriting attributes or properties of parent or base class. The derived class besides inheriting properties from the base class, can also have additional properties specified. The generic class of cars can be defined as an automobile that has an engine, 4 wheels, chassis and body. Specialized class of cars such as the Ford Taurus also have engine, 4 wheels, a chassis and a body. However the type of engine (in-line or V type), the type of tires (radial or normal), lighter chassis and body and airbags define the additional properties of this car. We can therefore say that the Ford Taurus has inherited the base class properties and also has additional new properties specified. Let us suppose that a customer requires a Ford Taurus with special seating arrangements. Ford will not start designing the Taurus from the beginning to implement the requirement. Instead, the feature that requires change is redesigned to suit the needs. This example shows that the custom seating

arrangement Taurus is an extension of the standard Taurus model. Thus, Ford designers are able to **re-use** earlier designs for implementing portions of the current design. Inheritance allows one to continuously **build** and **extend** classes from classes developed earlier.

Thinking in terms of objects of the real world allows us to re-use, and extend the object for future use. Object-oriented software programming therefore aims at re-using, and extending the modules of the program since changes are not expected in the old object, classes and methods, and most consist of extending the object's classes and methods.

In the real world the class of four wheeled automobiles can be defined as shown in

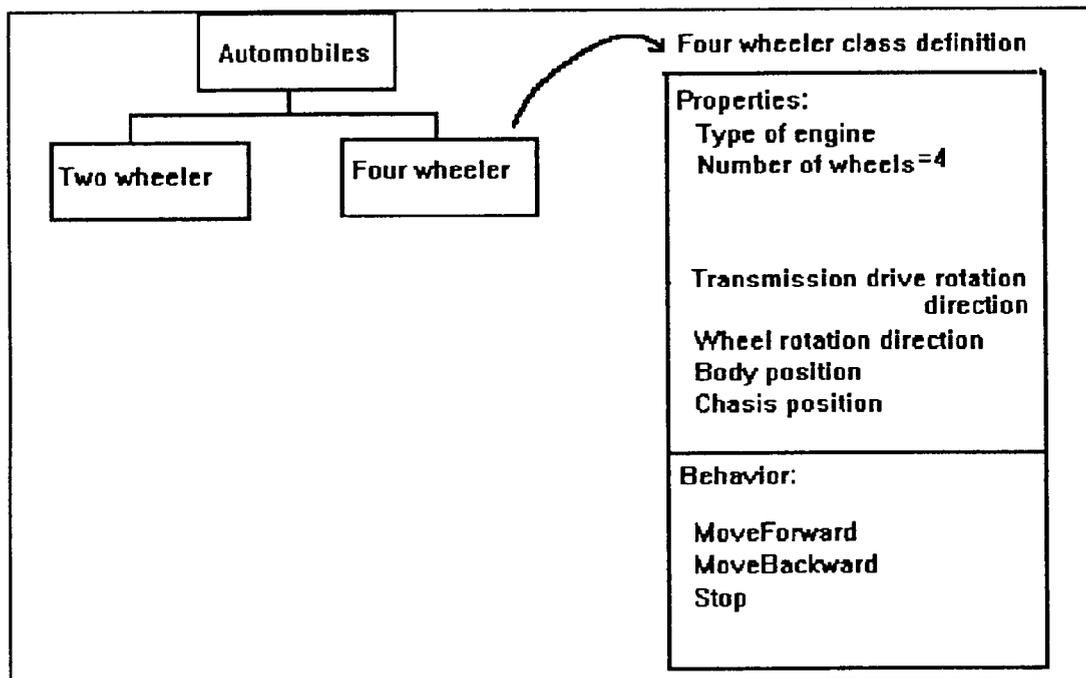


Figure 3.2: Real world definition of a four wheeled automobile

Figure [3.2]. In the definition of the class of four wheeled automobiles, the "moveforward" behavioral message causes the transmission drive and the wheels to rotate in a direction that causes the body and the chassis to move in the forward direction. Movebackward works in a similar manner. Stop causes the transmission drive to disconnect power transmission to the wheels causing the body and chassis to remain stationary. We see that the behavioral messages modify or act on the properties of the automobile. While this is definition of the real world object "automobile" as defined by a layman with no information about object-oriented analysis, the object oriented software programmer would define the same class of automobiles in C++ as:

```

class Automobile {
protected:
    char Type_of_engine; //Type of engine
    int  Num_Of_Wheels = 4; //Number of wheels
    BOOL Drive_Direction = 0; // Transmission drive direction
    BOOL Wheel_Direction =0; // Wheel rotation direction
    Point BodyPosition = 0; // Body position
    Point ChasisPosition=0; // Chasis position
public:
    Automobile ( ); // Constructor
    ~Automobile ( ); // Destructor
    virtual BOOL MoveForward ( ); // Behavior
    virtual BOOL MoveBackward ( ); // Behavior
    virtual BOOL Stop ( ); // Behavior
};

```

As can be seen from the definitions of the class of automobiles as defined by a layman and by the object-oriented software programmer, object-oriented programming allows the programmer to select real world objects, model their behavior in software so that we will achieve what the system has to do, and place them in class hierarchies according to their natural groupings.

In object-oriented programming with C++, the definition for any class is as shown above with the keyword **class** followed by the name of the class. The body of the class is delimited by braces and a semicolon. Faison [Faison 91] defines the responsibility of a class as :

The job of a class is to hide as much information as possible. Thus it is necessary to impose certain restrictions on the way a class can be manipulated, and how the data and the code inside the class can be used. There are three kinds of users of a class namely the class itself, generic users, and derived classes. Each kind of user has different access privileges. Each level of privilege is associated with a **keyword** and since there are three levels, there are three keywords; **private**, **public**, and **protected**.

The **private** keyword offers the strictest control allowing access to only the class of which the data is the member. Even derived classes do not have access to the private data members of their parent class. Accessing the member data(member functions or both) requires that they be declared in the **public** section of the class declaration. Data or functions declared in the public section allow access to anyone thus sometimes questioning the very integrity of data hiding. Classes that are used as base classes which allow the functions of the derived classes to access the data members of the base classes' employ the **protected** mode of data hiding restricting access to other classes.

In traditional programming, declaration of a variable of a built-in data type, such as **float** or **char** causes the compiler to automatically allocate enough space for that variable. In other words, the compiler *constructs* the variable by allocating the required memory and *initializing* the variable with a value. In a similar manner when a built-in data type goes out of scope, the compiler automatically returns (destroys) the memory associated with the variable.

When creating new objects, data members are un-initialized. C++ provides two special member functions to automatically initialize an object when it is created and destroy objects just before they go out of scope. They are:

* Constructor

* Destructor

The constructor has the same function identifier as the name of the class as shown below for the example on class.

```
Automobile ( );           // Constructor
```

A constructor is always called when an object is created, if a constructor is defined for the class. Constructors may be used to instantiate an object with default initialization, or specific initialization or by copying other objects. Constructors are normally declared public allowing generic class users to create objects from that class. An object, Example_Automobile, of the class Automobile may now be initialized dynamically as:

```
Automobile *Example_Automobile = new Example ( );
```

The destructor function is called to carry out operations that are to be performed when an object is no longer in use. Destructors are also normally declared public so that they can be used by generic class users. The destructor in the above example is called by invoking the member function:

```
~Automobile ( );       //Destructor
```

Let us now consider a small program written in Borland C++.

```

class CLWnd : public TWindow {
protected:
    BOOL lftmousebtndown; //Member data
    BOOL Draw; //Member data
    BOOL Erase; //Member data
    TPen *pen; //Member data
    TClientDC *lineDC; //Member data
    TPoint BeginPt, EndPt; //Member data
public:
    CLWnd(TWindow *parent, const char far *title); //Constructor
    ~CLWnd ( ); //Destructor
protected:
    void EvLButtonDown(UINT, TPoint &point); //Member function
    void EvLButtonUp(UINT, TPoint &point); //Member function
    void EvMouseMove(UINT, TPoint &point); //Member function
    void EvLButtonDbtClk(UINT, TPoint& point); //Member function
};

```

The code shown above is the definition of the class CLWnd and has been publicly derived from the TWindow class defined in Object Windows Library (discussed in section 3.3) of Borland C++. The constructor for the class is defined as:

```

CLWnd :: CLWnd(TWindow *parent, const char far
*title):TWindow(parent, title){
    lftmousebtndown = FALSE;
    Draw =FALSE;
    Erase=FALSE;
    pen = new TPen(RGB(0,0,0), 1, PS_SOLID); // Instantiation
    lineDC = new TClientDC(HWindow); // Instantiation
}

```

Since the CLWnd object instantiated the object dynamically and created a block of memory on the heap to hold an object "pen" of the type TPen with:

```
TPen *pen = new TPen (RGB(0,0,0),PS_SOLID);
```

A constructor that will free the memory on the heap when the pointer to the pen goes out of scope is then required. Therefore, the constructor for CLWnd is defined as:

```
~CLWnd :: CLWnd ( ) {
    delete lineDC;
    delete pen;
}
```

where the `delete` keyword frees the memory on the heap. In the above example, the CLWnd class has been publicly derived from the base TWindow class. While basic properties of the TWindow were inherited, new data members such as:

```
BOOL lftmousebtndown;
BOOL Draw;
BOOL Erase;
TPen *pen;
TClientDC *lineDC;
TPoint BeginPt, EndPt;
```

were added. This procedure of feature inheritance in object-oriented programming is described by Figure [3.3].

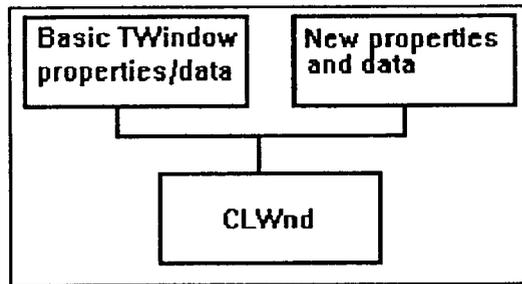


Figure 3.3: Multiple inheritance

Object-oriented programming also supports multiple inheritance where-in a class can be derived from more than one base class allowing access to the public members of base classes.

Inheritance allows the derived class to inherit the template of all the data members and the ability to call all of the non-private members of the base class. C++ also

```

class TObject{
private:
    int ObjType;
public:
    TObject(){
    TObject(int type){ObjType = type;}
    virtual int GetObjType( ){return ObjType;}
    virtual void SetObjType(int type){ ObjType = type;}
    virtual void BoundingRect(TPoint, TPoint) { }
    virtual double Distance(TPoint Point){}
    virtual BOOL Contains(TPoint Point){}
    virtual void HiLite(TDC &DevCon){}
    virtual void Modify(TPoint){}
    virtual void Draw(TDC &DevCon);
};
  
```

supports dynamic binding through the mechanism of virtual functions. A virtual function is a special member function that is invoked through a public base class pointer. When a non-virtual member function is called, the compiler decides at compile time which particular function to call. In contrast, everytime a virtual function is called, the executable code decides at run-time, during the call, which version of the virtual function to call. The selection of the code to be executed when a function is called through a pointer is delayed until run-time. The code that is executed is determined by the class type of the actual object addressed by the pointer or reference.

The code shown above is the base (parent) class of drawing objects (lines, circles, arcs *etc.*). The line object, TLine, is derived from this base class as follows:

```
class TLine: public TObject{
public:
    TPoint FirstCorn, SecondCorn;
public:
    TLine(TPoint S,TPoint E): TObject(LINE){ FirstCorn = S;
        SecondCorn = E; SetObjType(LINE);}
    virtual void BoundingRect(TPoint TopLeft,TPoint BottomRight){
        FirstCorn=TopLeft; SecondCorn = BottomRight;}
    virtual double Distance(TPoint Point);
    virtual BOOL Contains(TPoint Point);
    virtual void HiLite(TDC &DevCon);
    virtual void Modify(TPoint SetPoint);
    virtual void Draw(TDC &DevCon);
};
```

The virtual functions of the base class are over-riden in the derived class TLine. For example, the *virtual void Draw (TDC &DevCon)* member function of TLine over-rides the *virtual void Draw (TDC &DevCon)* member function of TObject as shown below.

```
//Member function of TObject
void TObject::Draw(TDC &DevCon){
}

//Member function of TLine
void TLine::Draw(TDC &DevCon){
    DevCon.MoveTo(FirstCorn);
    DevCon.LineTo(SecondCorn);
}
```

Although this chapter has covered the necessary and basic concepts of object-oriented programming, it is advised that the readers refer to the bibliography for detailed information on object-oriented programming.

Developing applications for Windows requires that one know Microsoft Windows programming methodology. Since Microsoft Windows is based on the concept of event based programming, the next section discusses the event-based Windows programming methodology.

3.2 Windows programming concept

This section discusses the advantages of implementing applications for the Microsoft Windows environment. This section also presents the concepts and implementation details of the event-based Microsoft Windows programming methodology.

3.21 Microsoft Windows

Microsoft Windows provides a consistent windowing and menu structure for all its applications. This makes Microsoft Windows based software packages easy to learn and use. Just as X-Windows is the graphical user interface environment in UNIX,

Microsoft Windows is a graphical user interface environment for MS-DOS. Microsoft Windows supports the popular *what you see is what you get (WYSIWYG)* feature. This visual environment speeds communication between users and computers. Since the user interface and tools are standardized, it allows the developer to concentrate on the functionality of the application rather than on the graphical user interface (GUI) for the same.

Programming for Windows ensures compatibility with a large number of input and output devices. Applications developed on Microsoft Windows need no special drivers as Microsoft Windows supports a host of display and printer drivers. Therefore, Microsoft Windows is said to support *device independent graphics*. Device independent graphics means that the application being developed is independent of the system configuration.

Microsoft Windows also supports *virtual memory management*, thus eliminating the DOS memory limitation of 640 K Bytes. The program can now access all the installed memory and any available virtual memory. With virtual memory management, segments of the program are run and swapped between the central main memory and the disk storage, thus allowing larger programs to run. Microsoft Windows supports multi-tasking, allowing the user to run multiple windows application.

All these advantages would encourage a non Windows programmer to shift towards Windows Programming. But as McCord [McCord 92] puts it...." Windows programming can be, quite honestly, a major headache to newcomers. Most programmers who are familiar with the traditional, sequential programming methodologies used with C are often not mentally prepared to meet the challenges of Windows programming."

3.22 Event-based programming

Windows programming is based on the event-based programming methodology as opposed to the traditional sequential programming method. It may then be questioned as to what is the difference between Windows programming and sequential programming. In Windows programming, an event such as a clicking of a button or selection of a menu item generates a message. Windows programming responds to these generated messages. Since these messages are non sequential and are event-driven, Windows programming methodology is termed as event-based programming.

Every Microsoft Windows application consists of a *WinMain* function. The *WinMain* function initializes the application, setting up the message loop. The initialization process consists of setting up of the parent window. The message loop consists of a system where-in all the input messages are directed to the system queue, which are then copied by Windows to the appropriate application queue. As Lafore [Lafore 93] puts it .." An application never does anything until Windows sends it a message".

In Microsoft Windows applications developed with *Object Windows Library* (OWL), the development kit from Borland, the TApplication derived class does the job of initializing the application and setting up the message loop as is defined in the OwlMain loop below. The *Run* member function of the TApplication initializes the first instance of the application and then the other instances of the same application.

```
int OwlMain (int,char *[ ]){
    return TDCSApp().Run();
}
```

After initialization, *InitMainWindow* is called to create the main window as shown below.

```
void TDCSApp::InitMainWindow(){
    //Construct the client "Canvas" window
    TWindow *client = new TCanvas(0,0);
    //Construct the main window "frame"
    TDecoratedFrame *frame = new TMainWnd(0,"DCS",client,TRUE);
    //Set MainWindow to frame
    SetMainWindow(frame);
}
```

Lastly, the Run member function calls the *MessageLoop* that receives and processes Windows messages sent to the application. In the code shown above, TDCSApp is the publicly derived class from the TApplication class of OWL.

```
class TDCSApp: public TApplication {
public:
    TDCSApp():TApplication(){}
    void InitMainWindow();
};
```

The application's message loop then retrieves the message from the application queue and sends each message to the appropriate window function.

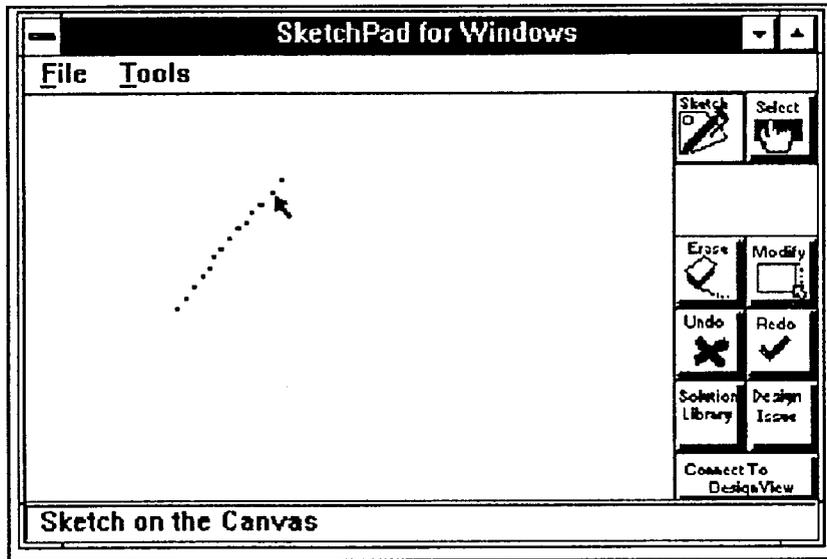


Figure 3.4: Input messages

Windows sends input messages when an application receives input from the mouse, keyboard, scrollbars, or system timer. Therefore, when the user presses the left mouse button down in the drawing area as shown in Figure [3.4], the *WndProc* function of Windows sorts the messages by processing information such as the window that generated the message, the message number (that is defined in *windows.h*), and additional information and sends the *WM_LBUTTONDOWN* windows message to the SketchPad application. Since the client window, the drawing area, was where the mouse button was pressed, the *MessageLoop* receives the Windows message, processes it and sends it to the client window. The process can be described as shown in Figure [3.5].

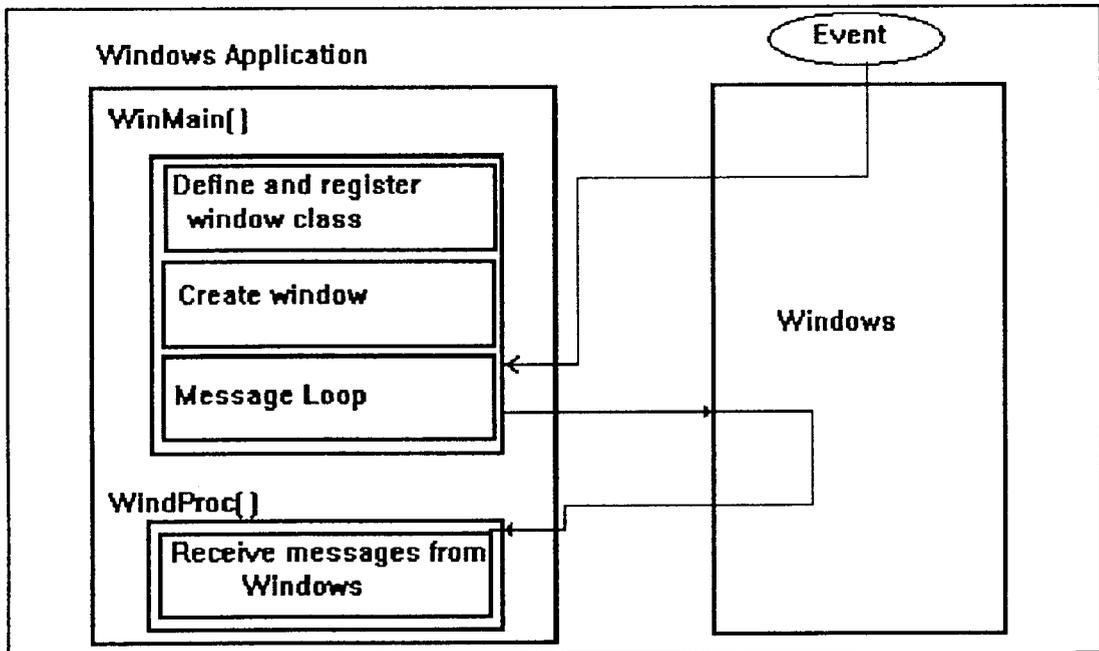


Figure 3.5: Windows Operation [Lafore 93]

Object Windows Library (OWL 2.0), discussed in section 3.3, offers an organized method of handling the above mentioned Windows events and messages. The event handler macro defined in the `RESPONSE_TABLE` for OWL classes, handles the message and breaks the `wParam` (unsigned integer-Two bytes) and the `lParam` (long - Four bytes) into separate parts. One might wonder as to what the `WM_` characters in the message `WM_LBUTTONDOWN` mean. The `WM_` term means that it is a Window Message. Some of the typical input Windows Messages and their meaning are shown in the table below. The messages listed are just a few from over 100 Window messages that Microsoft Windows supports.

Constant	Meaning
WM_LBUTTONDOWN	Left mouse button was pressed
WM_RBUTTONDOWN	Right mouse button was pressed
WM_DESTROY	Destroy a Window
WM_PAINT	Window needs to be redrawn
WM_MOVE	Window was moved

Table 3.1: Window messages

Message can be sent or posted. *Sending* a message means that Windows waits till the message is executed. *Posting* a message means that Windows posts the message to the message queue and does not wait till it is executed. For example, in the *SketchPad for Windows* application, the *CmSketch* member function of the *TMainWnd* shown below sends a message when the sketch button is clicked.

```
void TMainWnd::CmSketch(WPARAM Id){
    ClientWnd->SendMessage(PM_STATUS,Id,0);
}
```

In the code above, the number of the button clicked is sent as a number to the client window (*TCanvas*) which responds accordingly. When the user selects any menu item, Windows sends a *WM_COMMAND* message to the application's window procedure. Macros have been built in Borland C++ 4.0 to handle these command messages. For example, the *EV_COMMAND_AND_ID* macro takes the command

identification number and associates the function to be called when that command message is generated as shown below.

```
EV_COMMAND_AND_ID(CM_SKETCH,CmSketch),
```

where the `CM_SKETCH` has a defined identity number and is specified in the `.RC` file and `CmSketch` is the member function associated with it..

Since Microsoft Windows provides a consistent windowing and menu structure for all its applications, it supports Application Program Interface functions, commonly known as API. There are more than 600 Windows API functions available for use. These Windows API functions allow Windows Applications to manage window controls, memory, graphics utilities, data input and output. For example, the message box that pops-up in Windows applications, is an API function. Similarly, the graphics functions such as `LineTo`, `MoveTo`, `Ellipse` etc. are all examples of the API function.

Borland gives the application developer an encapsulated version of the API functions, thus making programming with Windows easier. These encapsulated functions are provided in the application frame work called Object Windows Library (which will be discussed in the next section).

3.3 Object Windows Library (OWL 2.0)

This section covers the implementation of Window elements (all major elements, including windows, dialog boxes and interface objects) as objects with defined behaviors, attributes and data. A thorough understanding of this section is necessary to write working applications for Microsoft Windows.

Object Windows Library is an application framework developed by Borland that incorporates the advantage of object-oriented programming for writing Windows applications. OWL runs in graphics mode under Windows and hides the low-level details from the application code. Walnum [Walnum 94] describes programming with OWL as... "Using OWL, one can create a fully operational window in eight or fewer lines of code while the same when written in C (without OWL) requires more than 80 lines to produce the same window".

The OWL library encapsulates window information, abstracts Windows functions and takes care of automatic message response. In other words, Object Windows defines the behavior, attributes and data while Windows implements the physical representation of the objects. Object Windows makes use of member functions that abstract Windows functions through built-in interfaces. In Object Windows, a member function can be defined for each message that must be handled so that when the object receives the message, the appropriate member function is called automatically.

Object Windows includes many classes with which one could write a Windows Application quickly and easily. Window objects such as the basic window, dialog boxes, buttons, menu bars, status bars *etc.*, are instantiated from the classes defined in *Object Windows Library*.

Some of the important classes that are included in the Borland 4.0 version are:

- * Application
- * Windows
- * Menus
- * Dialog Boxes
- * Child Controls
- * MessageBar, SpeedBar

Applications, such as *SketchPad for Windows*, written with OWL need to support at least two kinds of objects: An application object and a window object. The application is the framework that manages all the objects in the programs, sets the

application's message loop running so that the application can interact with the user and Windows.

Object Windows is so powerful that a fully functional window (that which allows minimization, maximizing, resizing and moving) can be generated in 5 lines.

The code shown below generates a fully functional window as shown in Figure [3.6].

```
#include <owl\application.h> //Line1
int OwlMain(int, char*[]){ //Line2
TApplication Example("TApplication Example"); //Line3
return Example.Run( ); //Line4
} //Line5
```

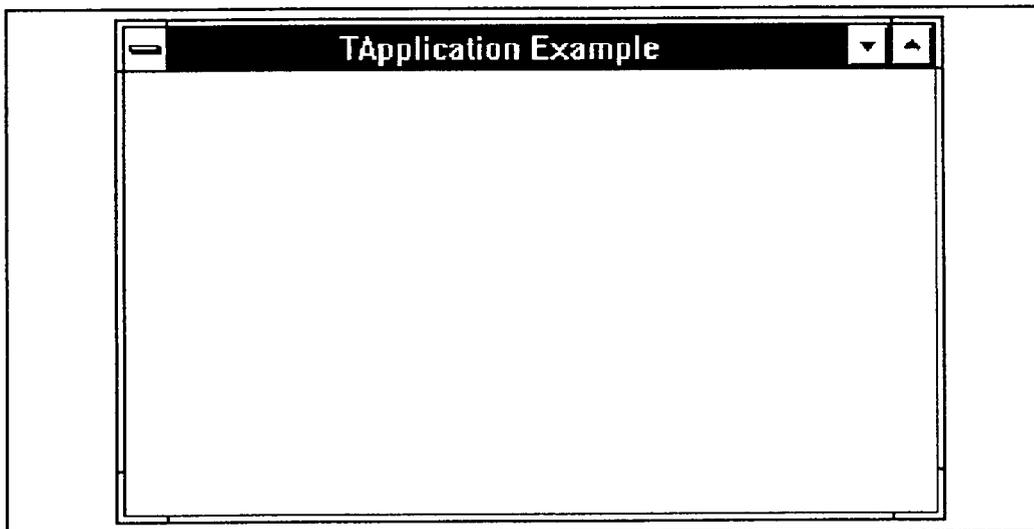


Figure 3.6: Basic OWL Window

For every OWL object used we need to include the relevant header file. Line 2 is where the OWL application starts execution. In line 3 we make an instance of the TApplication class by using the single argument constructor of TApplication. Line 4 sets the application object Example running, thus initializing the application and establishing a message loop.

The Window that is created by this example is not very impressive and needs to be customized for one's application by using the window classes. A closer look into the TApplication InitMainWindow function reveals that the application starts up a bare TWindow window object. Instead of the bare window object we can choose to use Frame Windows, Decorated Frame Windows, and Multiple Document Interface (MDI) Windows.

Let us suppose that we wished to implement a Frame Window, instead of the default TWindow, that responded with a message when the user clicked anywhere in the window. It can be done as shown in the code below. The application is as shown in Figure [3.7].

```
#include <owl\application.h>
#include <owl\framewin.h>
class Example : public TApplication {
public:
    Example ( ):TApplication ( );
    void InitMainWindow ( );
};
class ExampleWindow : public TFrameWindow {
public:
    ExampleWindow(TWindow *parent, const char far *title) :
        TFrameWindow(parent,title);
protected:
    void EvLButtonDown(UINT, TPoint &point);
    DECLARE_RESPONSE_TABLE(ExampleWindow);
};
```

```

DEFINE_RESPONSE_TABLE1(ExampleWindow, TFrameWindow)
    EV_WM_LBUTTONDOWN,
END_RESPONSE_TABLE;
//Member function of the TExample class
void Example:: InitMainWindow ( ){
    ExampleWindow *window = new ExampleWindow (0, "New Window");
}
//Member function of the TExampleWindow class
void EvLButtonDown (UINT, TPoint &point){
    MessageBox("Left Mouse Button was Clicked", "Example", MB_OK);
}
//Main Loop
int OwlMain( int, char* [ ] ){
    return Example.Run ( );
}

```

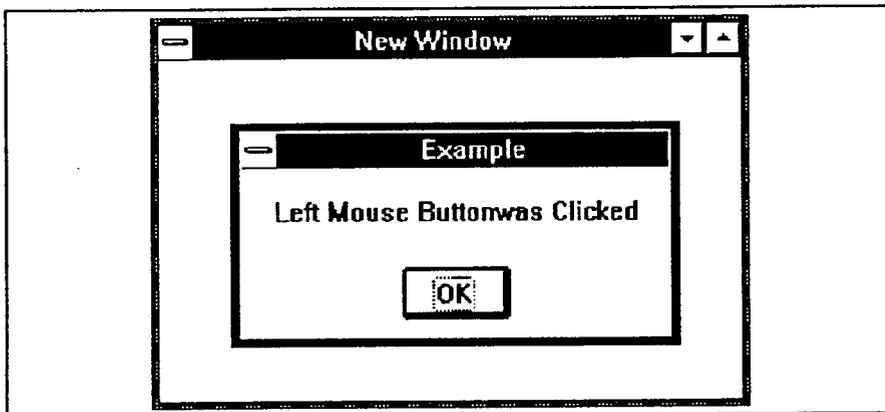


Figure 3.7: Improved OWL Window

When the user clicks the left mouse button anywhere in the window, a message box

appears, informing the user that the left mouse button was clicked. The function *MessageBox* displays a window containing a specified caption and text on the screen. Since a Windows Application is event driven, everything that the user does generates an event which the application receives in the form of a Windows Message. In this example the application responds to the Windows message WM_LBUTTONDOWN.

OWL uses message-response functions for handling the Windows messages. The response table shown in the code above matches the message-response functions (EvLButtonDown) with the messages (WM_LBUTTONDOWN) that they are supposed to handle. Some of the message-response functions of OWL 2.0 and the Windows messages that they handle are shown below.

Message-Response Function	Windows Message
EvLButtonDown	WM_LBUTTONDOWN
EvLButtonUp	WM_LBUTTONUP
EvMouseMove	WM_MOUSEMOVE
EvPaint	WM_PAINT

Table 3.2: Response functions and Window messages

The Windows message WM_LBUTTONDOWN comprises of **wParam** and **lParam** parameters. While the former parameter indicates whether the virtual keys are down, the latter parameter contains the coordinates of the cursor expressed relative to the upper-left corner of the window. The Borland message-response function

EvLButtonDown automatically extracts the virtual key flag as an unsigned integer (UINT) and the coordinates as a TPoint object thus making Windows programming easier. If instead of the *MessageBox* popping up, the text "Left mouse button was clicked here" be displayed , then only that portion of the code contained in the EvLButtonDown function needs to be changed as shown below. While employing the OWL version of the Windows *TextOut* function, the header file *dc.h* should be included in the source code.

```
void ExampleWindow :: EvLButtonDown ( UINT, TPoint &point){
    TClientDC* DContext = new TClientDC (HWindow);
    DContext->TextOut(point, "Left mouse button was clicked here");
    delete DContext;
}
```

Now that we have implemented the *Window* classes we shall move on to the implementation of menus. When a user loads a new Windows application, he or she may not be able to make full use of the application but will at least know where to locate the File operations such as opening a new file, opening an existing file, saving a file. This consistency in all Windows applications is largely due to the specifications or guidelines laid down for Windows application development. We shall see how the File menu can be implemented.

The Resource Workshop (explained in section 3.4) from Borland enables the user to create and test menus. The menu editor consists of the outline pane, the test menu pane, and the dialog box pane. Editing the default menu from the outline pane with the help of the dialog box pane and by selecting the " New File popup item" from the Menu menu, the basic File menu can be implemented without any effort. The Resource Workshop 4.0 allows the user to specify and verify the message that the

individual items in the menu generate. This menu is stored in the .RC file that is going to be used in the application. An example of a menu [Figure 3.8] and its .RC file is shown below.

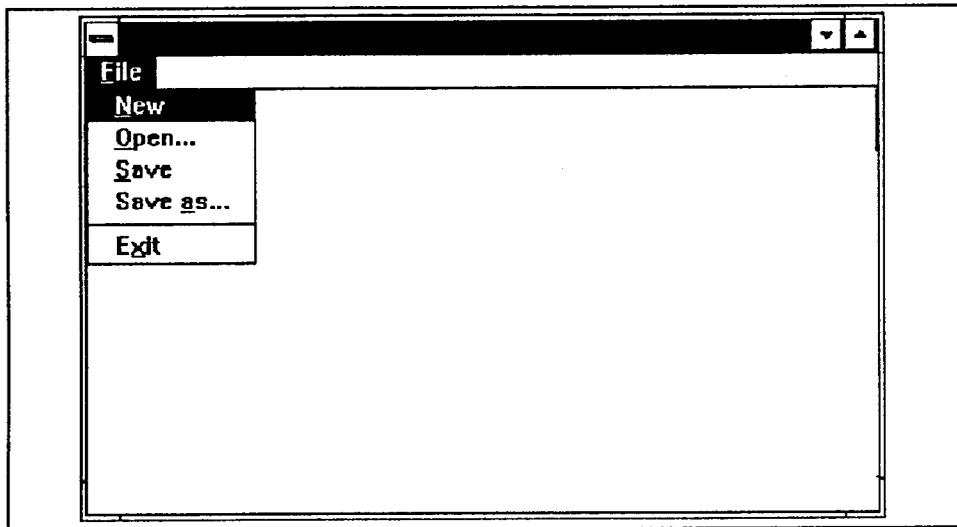


Figure 3.8: Menu implementation

Selection of any of the menu commands like New or Open will cause a new window to come up or open up an existing window respectively. For example, selection of the New option from the menu causes the application to create a new window or document depending on the application it is designed for. Since Windows programming revolves around objects and the messages that they send to communicate with each other, a closer look at the code shown below reveals that the selection of the New option causes the command message (abbreviated *CM_message*) `CM_FILENEW` to be sent. Now the window class should include message-response functions for each menu item that has to be handled.

```

#ifndef WORKSHOP_INVOKED
#include "windows.h"
#endif

#define DCSMENU           1
#define CM_FILENEW       24331
#define CM_FILEOPEN      24332
#define CM_FILESAVE      24333
#define CM_FILESAVEAS   24334
#define CM_FILEEXIT     24338

#ifdef RC_INVOKED
EXAMPLEMENU MENU
{
  POPUP "&File"
  {
    MENUITEM "&New", CM_FILENEW
    MENUITEM "&Open", CM_FILEOPEN
    MENUITEM "&Save", CM_FILESAVE
    MENUITEM "&Save &As...", CM_FILESAVEAS
    MENUITEM SEPARATOR
    MENUITEM "&Exit", CM_FILEEXIT
  }
}
#endif

```

This is achieved by including the message-response functions in the window class and defining the response table for the window as shown below.

```

class ExampleWindow : public TFrameWindow {

public:
    ExampleWindow(TWindow *parent, const char far *title);
protected:
    void CmFileNew ( );
    void CmFileOpen( );
    void CmFileSave( );
    void CmFileSaveAs( );
    void CmFileExit( );

    DECLARE_RESPONSE_TABLE(ExampleWindow);
};

```

```

DEFINE_RESPONSE_TABLE1(ExampleWindow, TFrameWindow)
    EV_COMMAND(CM_FILENEW, CmFileNew),
    EV_COMMAND(CM_FILEOPEN, CmFileOpen),
    EV_COMMAND(CM_FILESAVE, CmFileSave),
    EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs),
    EV_COMMAND(CM_FILEEXIT, CmFileExit),
END_RESPONSE_TABLE;

```

The EV_COMMAND macro includes in its parenthesis the command message's ID and the name of the function that is to respond to that command. The command message's ID and the respective functions that are called are shown in the table below.

COMMAND MESSAGE'S ID	FUNCTION INVOKED
CM_FILENEW	CmFileNew ()
CM_FILEOPEN	CmFileOpen ()
CM_FILESAVE	CmFileSave ()
CM_FILESAVEAS	CmFileSaveAs ()
CM_FILEEXIT	CmFileExit ()

Table 3.3 : Command messages

In a similar manner all the menus required for developing the *SketchPad for Windows* application can be implemented. Once a Windows application developer gains some experience in writing Windows applications, he or she can develop the menu structures without the help of the Resource Workshop.

Anyone who has used a Windows application would have seen dialog boxes used extensively in the application. One of the most commonly seen dialog box is the one that popup when the user selects Open from the File menu [Figure 3.9]. Dialog boxes are used to open and save files, accept input and change colors etc. Dialog boxes contain child controls such as *pushbuttons*, *radiobuttons*, *checkbox*, *combination boxes*, *static text* etc. As in the case of development of the menu's for Windows application we employ the DialogBox resource editor of the Resource Workshop (see section on ResourceWorkshop) to develop the required dialog box . Dialog boxes may be modal (these do not allow the user to utilize the dialog box's parent window when the box is active), modeless (these allow the user to utilize the dialog box's parent window even when the box is active) or system-modal (these are used to disable access to other windows until the user responds to the dialog box).

The *STYLE* parameter of the dialog box defines the type of dialog box. The *CONTROL* parameter of the dialog box defines the type of child control used.

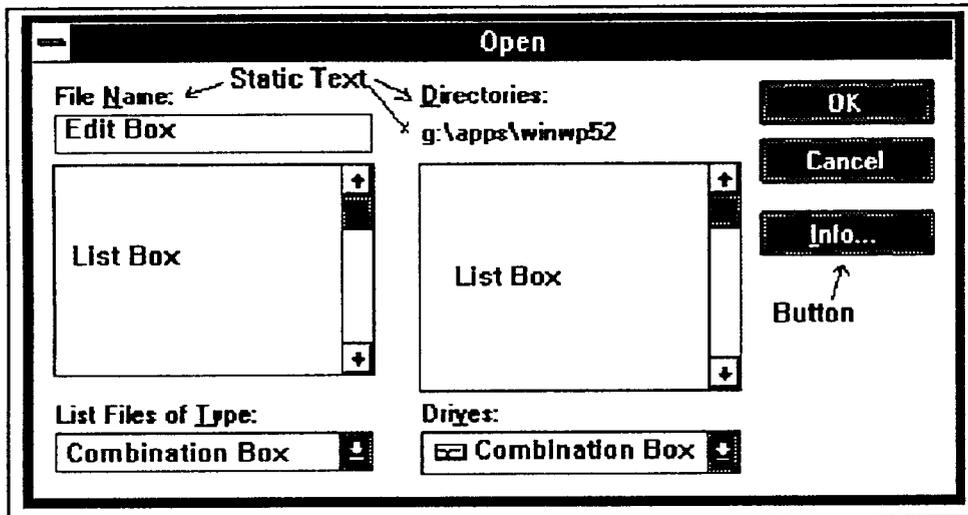


Figure 3.9: Open/Save Dialog Box

Buttons (pushbuttons, radio buttons, bitmapped buttons), editing controls, static text controls, scroll bars, list boxes, and combo boxes constitute the child controls *Object Windows Library*. The editing control (TEdit class) allows the user to edit the data. Examples of editing control implementation can be found in the Find or Search dialog boxes implemented on commercial applications. Scroll bars (TScrollBar class) are extensively used where data exceeds the display and needs a positioning device to view the entire data. List boxes (TListBox class) provide a list of options from which the user can choose an item. If too many items in the list are to be displayed at once, a scroll bar is displayed automatically.

```

#ifndef WORKSHOP_INVOKED
#include "windows.h"
#endif

#define DIALOGBOX_EXAMPLE 1
#ifdef RC_INVOKED

DIALOGBOX_EXAMPLE DIALOG 11,23, 174,124
STYLE
DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_S
YSMENU
CLASS "bordlg"
CAPTION "Example Dialog Box"
FONT 6, "Times New Roman"
{
CONTROL "",
ID_OK,"BorBtn",BS_PUSHBUTTON|WS_CHILD|WS_VISIBLE|WS_T
ABSTOP, 66,91,37,25
}
#endif

```

An example of the list box and the scroll bar is the file selection option in the File Save or Open dialog box . Combination boxes (TCombination class) are commonly seen in the font selection dialog box where-in the default size is displayed beneath which a list box containing different font sizes is displayed. Buttons are the most commonly used form of child controls. Buttons are widely used to create speedbar buttons, and to accept user's decisions such as acceptance, cancellation *etc.*,.

Implementing the window controls on dialog box is a lot more easier, thanks to the powerful resource workshop. Child controls are inserted on dialog boxes with the CONTROL parameter as shown earlier in the dialog box example. Implementing window controls on windows is however done in a different manner. Here the control's parent-window pointer and its ID and also its exact position in the window

should be supplied. For example, a TListBox object and a TRadiobutton object, can be implemented as shown in the code below.

```

class ExampleWindow : public TFrameWindow {
protected:
    TRadioButton *Button1;
    TListBox     *Listbox;
public:
    ExampleWindow (TWindow *parent, const char far *title);
};

ExampleWindow::ExampleWindow ( TWindow *parent, const char far*title){
    Button1 = newTRadioButton (this, ID_RADIOBUTTON, "RB", 20,20,50,20);
    ListBox = new TListBox (this, ID_LISTBOX, 120, 20, 100,50);
}

```

Borland has incorporated lots of useful and time saving features in version 4.0. For example working with button bars or status bars or message bars was a tricky affair in the earlier versions. But in version 4.0, Borland has implemented classes for control bars, tool bars, message bars, and status bars. Most of the Windows applications have speed bars to speed up the basic file, editing and printing operations. An example of a speed bar is as shown in Figure [3.10]. Speedbars are called control bars in Borland C++ 4.0. A control bar can be implemented in a *Decorated Frame Window* by calling the TControlBar constructor. The TControlBar constructor takes a pointer to the parent window, a tile direction (the direction in which the control bar buttons should be placed), a pointer to a TGadgetWindowFont, and a pointer to TModule.

Since all these parameters have default values, which can be checked by looking at <owl\controlb.h> header file, only the pointer to the control bar's parent window needs to be given. The tile direction, vertical or horizontal can be specified as the second parameter. The control bar is therefore implemented as:

```
TControlbar *cntrlbar = new TControlBar(frame);
```

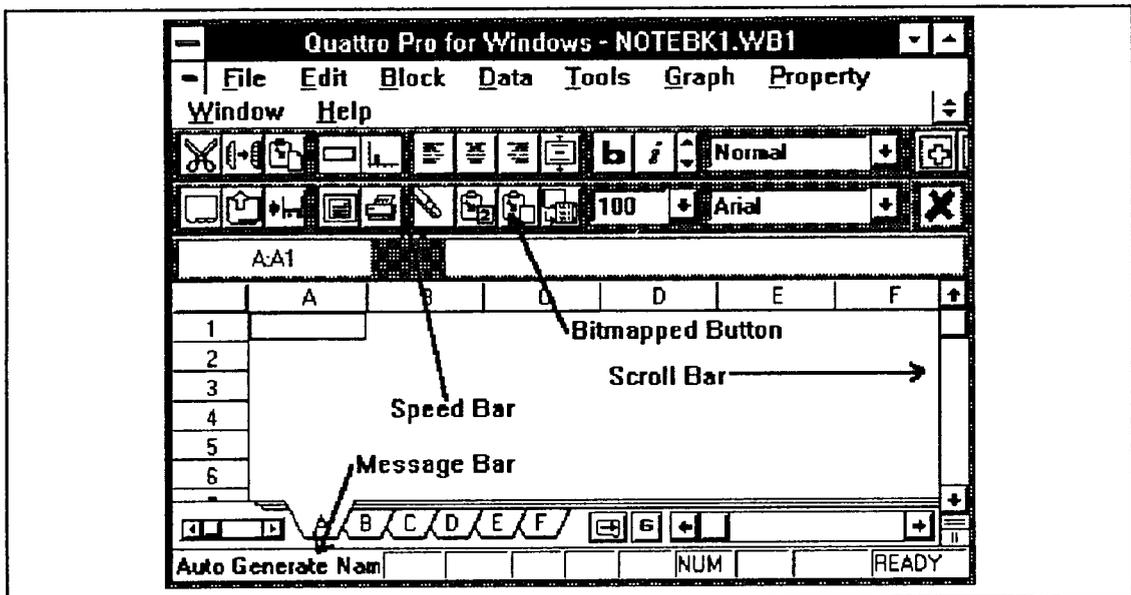


Figure 3.10: SpeedBars in applications

Buttons, or Button Gadgets in the Borland language, for the control bar can then be implemented by calling the `TButtonGadget` constructor which takes the resource ID of the bitmap representing the button's face, the menu command (or any command) that the button represents, the button type, a Boolean value indicating whether the button is enabled, the button's state, and a Boolean repeat value. Since we want the button to be a default push button and not an exclusive button, only the first two parameters need to be given as shown below.

```
cntrlbar->Insert( new ButtonGadget(BMP_OPEN,CM_FILEOPEN));
```

The *Insert* function takes care of adding the button to the control bar. Incidentally the *Insert* function takes a pointer to the new gadget. Control bars have gaps or separation in between some of the buttons. As the name suggests, separator gadgets (TSeparatorGadget class) are used to provide gaps or spaces in between button gadgets. A TSeparatorGadget is inserted in the manner the TButtonGadget is inserted in the control bar. The toolbox, message bar and status bar are all incorporated in a similar manner.

The program elements (buttons, dialog boxes etc) implemented in the source code, are represented visually by resources. The commonly used resources such as menus, dialog boxes, bitmaps, icons, cursors and string tables, their creation and visual representations are as explained in the next section on resources.

3.4 Resources and graphical user interface development

This section discusses the implementation of resources in order to impart the standard Microsoft Windows "look and feel" to the application being developed.

The most striking feature of Windows is the graphical user interface. This common graphical user interface is brought about through the use of standard Windows resources such as Windows, Menus, Dialog Boxes, Bitmaps, Icons, Cursors, and String Tables [Figure 3.11].

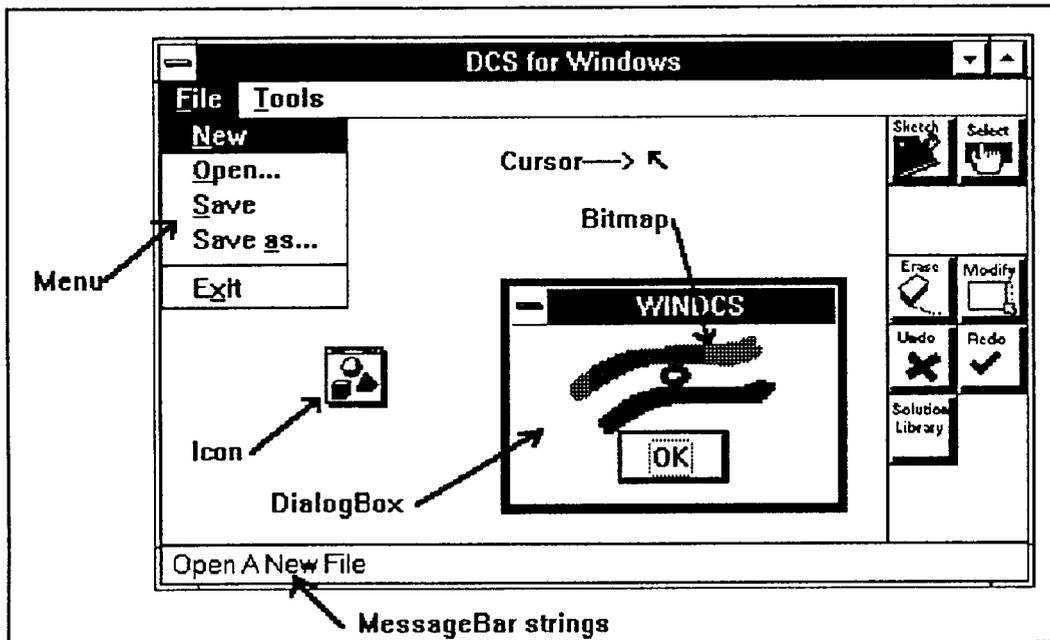


Figure 3.11: Windows Resources

The resource workshop allows the user to edit and create both binary and text files. The most commonly used format is the .RC format which stands for Resource Script format that contains one or more resource definitions. Other formats for storing bitmaps, fonts, cursor, etc are used but since all these formats can be used in one format, the .RC format is preferred for most projects. The Resource Workshop 4.0 allows the user to list and visualize the resources stored in the file [Figure 3.12].

Resources are merely the visual representation of the program elements implemented in the source code. Since the same resources can be used in many applications, it can be separated from the source code and bound with the application executable. Resources are created by the various editor and combined into a single file called resources. This resource file is then combined with the program's .EXE file to create a new improved .EXE file [Figure 3.13]. Resources take a lot of memory. To conserve memory, Windows waits until a resource is actually needed in the course of

the program execution before loading it.

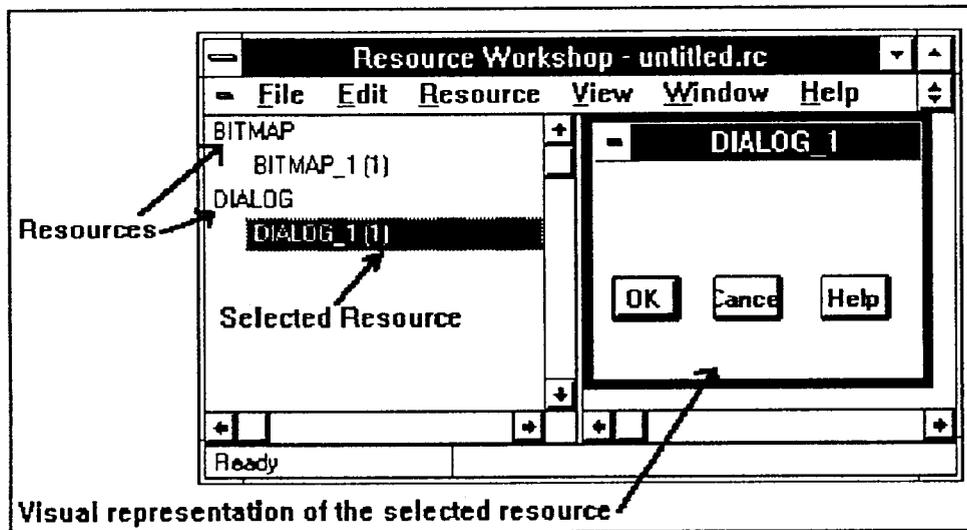


Figure 3.12: Visual representation of resources

Just as the menu at the restaurant helps one in deciding the cuisine to be ordered, software menus guide the user to look for a particular command option by wading through the hierarchy of menu. Gone are days when the user had to type all the commands. The menu editor of the Resource Workshop allows the user to create menus and dynamically check them too. The menu editor is as shown in Figure[3.14]. The menu editor is divided into three windows. One of the windows is the testing window (located in the upper right hand corner), the other is the display or outline window that contains the code for the menu and lastly the window for editing the menu features (such as graying, popup). The editing environment is dynamic in the sense that any changes made in the editing window are immediately reflected in the test window.

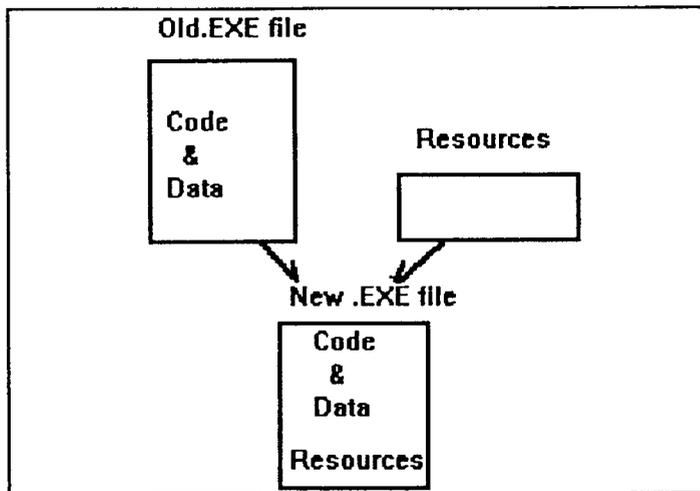


Figure 3.13: Resource binding[Lafore 93]

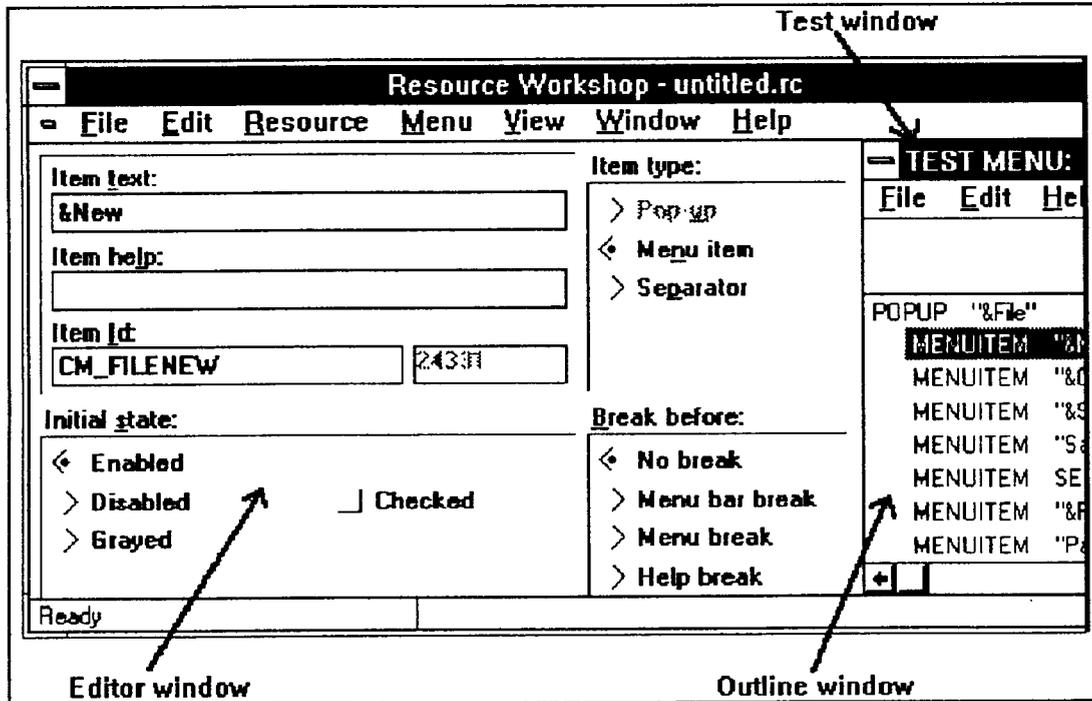


Figure 3.14: Menu Editor

Dialog Boxes, as the name suggests, promotes the exchange of ideas between the user and the program. The exchange of ideas takes place through a set of tools, called controls. Dialog Boxes are usually pop-up windows. While custom dialog boxes (Open File, Save File, Find and Search) are pre-designed and ready for use, custom dialog boxes (which look and work the way we want) can be prepared with the help of the dialog box resource editor [Figure 3.15].

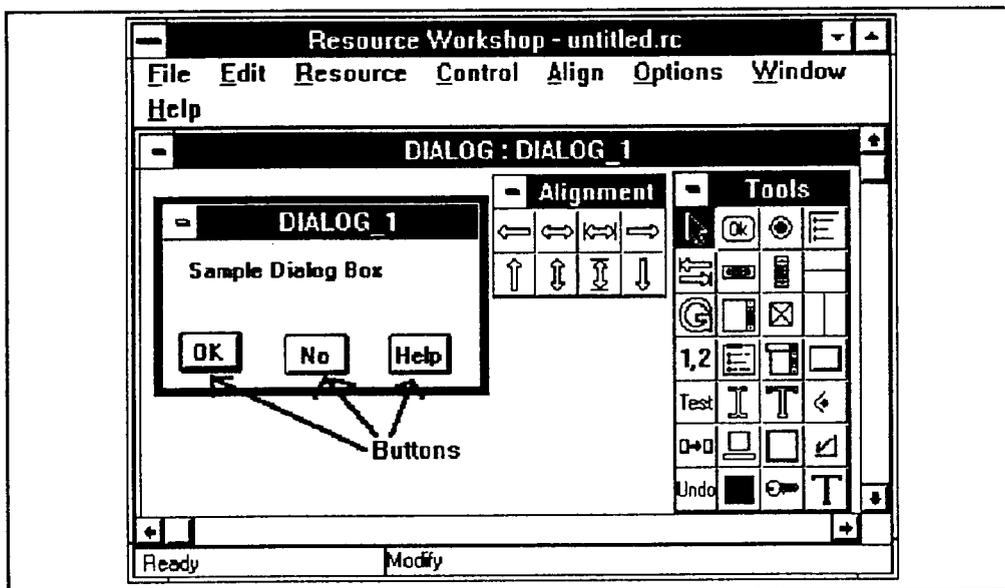


Figure 3.15: Dialog Box editor

The Borland Resource Workshop developer's have been adding features to this powerful tool-kit in order to provide more nearly realistic dialog resource features. The term "realistic" refers to the light gray, chiselled look features possessed by Motif dialog boxes for the X-Window environment. Depending on the dialog box selected, a default dialog box appears. A tool palette and an alignment palette also open up. While the Borland chiselled-look dialog boxes possess the default OK, CANCEL and HELP buttons, default standard dialog boxes appear with no child controls on them.

Child window controls, such as the information icons, bitmaps, buttons (pushbutton, radiobutton, bitmapped buttons), listboxes, groupboxes, checkboxes, static text, editing window, combination boxes, raised or dipped shades can be used alone or in combination in the design of custom dialog boxes.

Most Microsoft Windows users have used the PaintBrush application at least once. The PaintBrush application is basically a bitmap editor. Bitmaps are graphical images that one can use in the program. Bitmaps are images formed by a pattern of bits. Present day applications extensively use bitmapped buttons on the speed bar (menu bar). These buttons are generated by generating bitmaps. The bitmaps, although can be developed on PaintBrush, are developed on the bitmap editor from Borland [Figure 3.16].

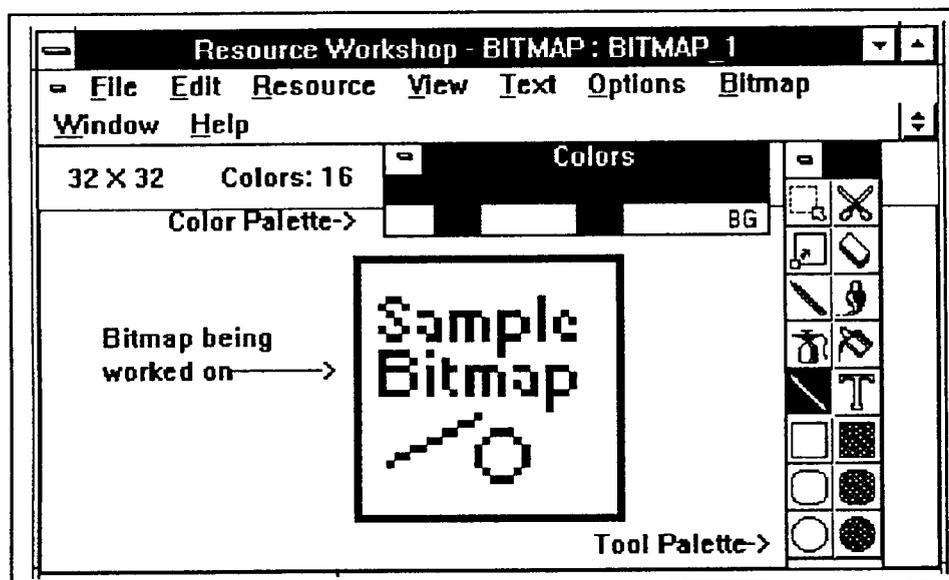


Figure 3.16: Bitmap editor

On being invoked, the bitmap editor, opens up a child window for painting. With the

help of the tool-palette and color-palette, one can generate the desired graphics and store them as bitmaps.

Upon starting MS Windows, the user is presented with the program manager window consisting of many images in the window. These images are nothing but bitmaps representing the particular application's minimized status. These bitmapped images of minimized windows are called icons. Icons are usually either 32-by-32 or 16-by-32 pixels. Since the icons are basically bitmaps, the icon editor works similar to the bitmap editor.

With the advent of graphical user interface (GUI), pointing resources such as cursors have been employed to locate the position of the mouse on the screen. Cursors are 32-by-32 pixel bitmaps and are commonly used to locate positions on the screen, select buttons, draw object and perform editing functions. Some of the commonly used cursors and their functionality are mentioned below.

Cursor Shape	Action or Use
Arrow	General
Cross Hairs	Selection, Drawing
Hour Glass	Wait Status

Table 3.4: Cursors and their use

As one wades through the menus, the message bar located at the bottom displays the information about each menu item. These messages are sections of text called strings.

Since they are stored outside the program, changing strings to accommodate for a foreign language string becomes easy. Strings are defined in the string table resource editor. The editor allows one to create and edit string tables and consists of three columns (ID source column, ID value column and the string column) [Figure 3.17]. Whenever an ID source value is called or referred to, the respective string is displayed in the message bar area.

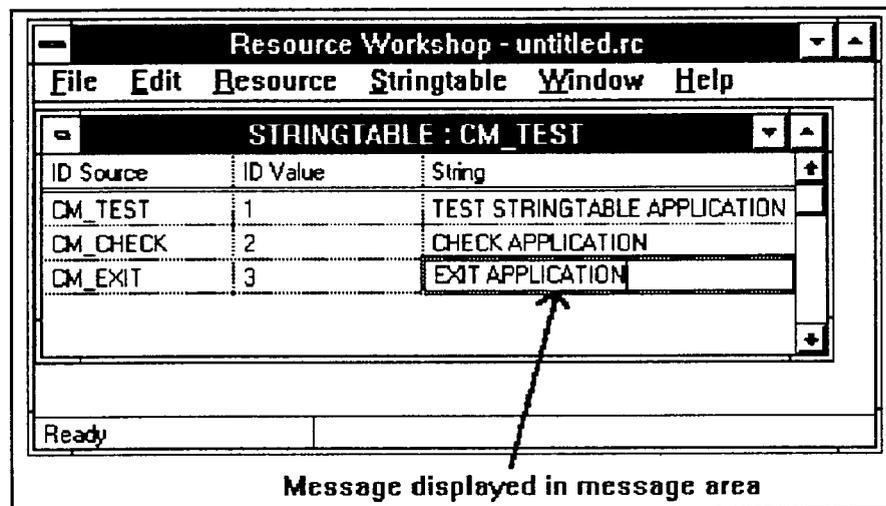


Figure 3.17: Stringtable editor

For further information on object-oriented programming, windows programming, and Object Windows Library programming, it is suggested that the reader refer to the bibliography at the end of this thesis.

Section 3.1 dealt with the basics for utilizing the OWL framework to develop the Microsoft Windows applications. Since this framework is object-oriented, section 3.1 dealt with the basics of object-oriented programming. Since programming methodology for Microsoft Windows is event-based, section 3.2 discussed this topic.

4.0 SketchPad for Windows

This chapter discusses the implementation of the sketching features in *SketchPad for Windows*. While section 4.2 explains the reasoning behind the implementation of features of inking and CAD systems into this sketch capture system, section 4.3 explains the working and implementation of the features of *SketchPad for Windows* with an example. The features demonstrated in section 4.3 are implemented in C++ with Borland's Object Windows Library (OWL 2.0) and the implementation details are explained in section 4.4.

4.1 Introduction

SketchPad for Windows is the beta version of the sketch capturing software developed at the department of mechanical engineering, Oregon State University. It was developed as a tool primarily to help the design engineer in the design process. *SketchPad for Windows* operates in the Microsoft Windows environment. *SketchPad for Windows* has been developed to support the sketching ideology explained in chapter 2. Based on the philosophy that the sketching process requires minimal use of instruments, *SketchPad for Windows* allows the design engineer to sketch on the computer screen just as he sketches on a piece of paper. The design engineer no longer has to express his/her thoughts in the CAD form, as *SketchPad for Windows* automatically stores the sketch in CAD format.

As mentioned earlier (see Chapter 1), *SketchPad for Windows* is more than a mere sketching application. It is a sketching tool linked with a design solution library, issue based information system and a parametric CAD software. It is aimed at working in unison with the *Solution Library* [Wood 95] and *Decision Support* [Herling *et al.* 94] system. The sketching application also supports an interface to DesignView, a parametric drafting software to generate detail drawings. The

integration of *SketchPad for Windows*, Decision Support system, Solution Library and the parametric CAD can be said to mimic the design process as the system allows the designer engineer to sketch designs, store and retrieve designs using functional behavior, compare other designs and finally generate detailed drawings. This integration can be represented as in Figure [4.1].

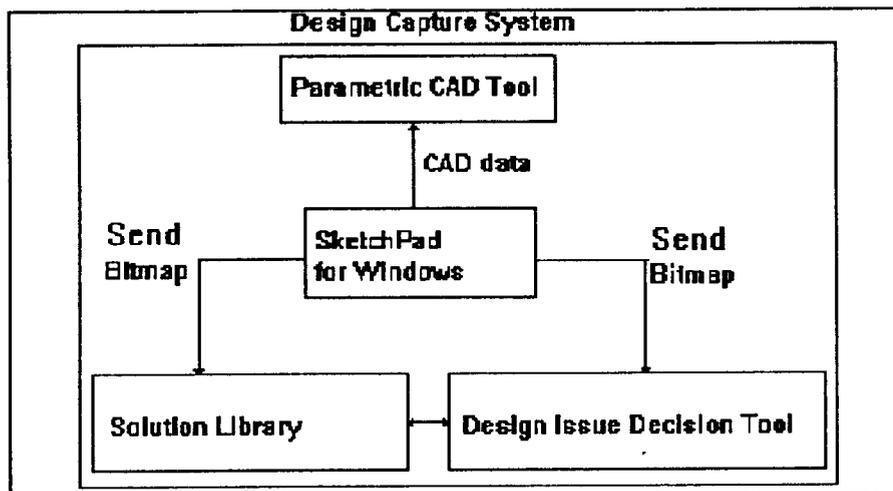


Figure 4.1: Design Capture System

As can be seen from Figure [4.1] and also from Figure [1.1], *SketchPad for Windows* communicates with the Solution Library and Decision Support system in the form of bitmaps while the sketches are converted into CAD data. Currently, *SketchPad for Windows* supports transfer of data into DesignView, a variational geometry CAD software.

4.2 Motivation for implementing SketchPad for Windows

In order to capture the traditional paper-and-pencil sketching environment on a computer it was necessary to study the traditional sketching procedure. In the

traditional sketching procedure the user is uncertain about the stroke that he makes. Lines are created by stroking from a point to another while constantly keeping one's focus on the other end point. Similarly, circles and ellipses are sketched by stroking sectors of the arcs constituting their profiles. Small circles and ellipses are however sketched in one stroke.

Current CAD software do not allow the user to draw or sketch drawing entities in the manner described in the previous paragraph. For example, drawing a line on a CAD software requires that the user select an icon for drawing lines, a start point, an end point and the software then draws a line between the two points. Thus, there is a difference in the way a designer sketches on paper than on a CAD system.

The computer graphics technology that closely matches the traditional sketching procedure is the inking process. A very common example of an inking application is the PaintBrush application for Microsoft Windows. PaintBrush for Microsoft Windows possess an inking feature that follows the position of the mouse as it moves across the screen [Figure 4.2].

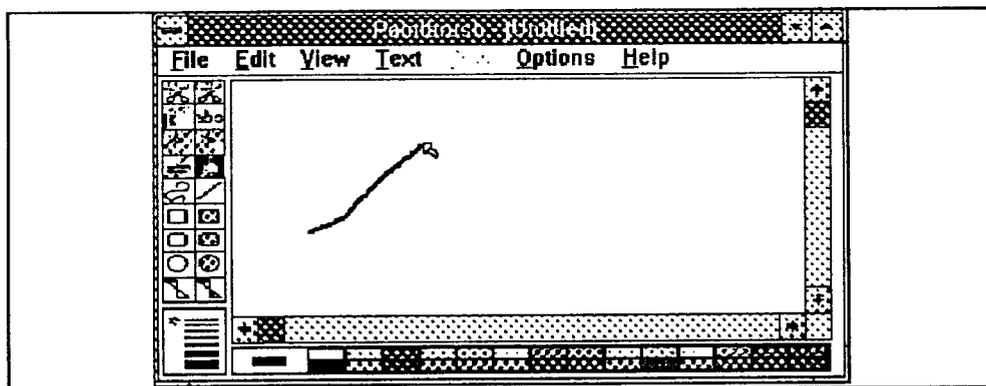


Figure 4.2 Inking process

Inking makes the cursor automatically leave a trail of line segments the way a pen leaves a trail of ink. It does not require that the user push the mouse button for every line segment, but instead draws a new segment whenever the cursor moves a sufficient distance.

A striking feature of the inking process is the fact that it does not require the user to think in terms of any menu or icon for thought representation, and allows the user to sketch freely on the canvas area of the application.

Consider that a design engineer makes a sketch using the above mentioned software. If the designer desires to modify the design or sketch, he/she must work on a bitmap and may have to resort to erasing the bitmap or redrawing the sketch. Sketch refinement is therefore hampered as the user must constantly re-sketch the original design. Bitmaps can however be cut, pasted, moved, and/or deleted.

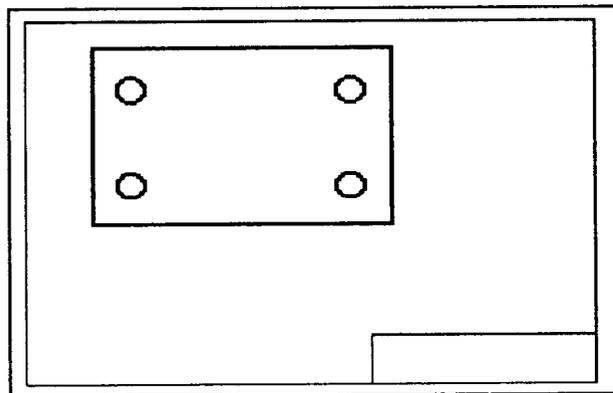


Figure 4.3: Drawing Information

Bitmaps require large amounts of memory as they contain bit-by-bit information of the entire drawing area Figure [4.3]. As shown in the bitmapped figure above, large

portions of the drawing area contain no information about the sketch. Storing information about these areas is of no use as they contain no drawing information.

Moreover, in an inking application, information about every pixel rather than that of the entity topology is stored. CAD software however store topology information rather than bit-by-bit information. Therefore, if the inking process could be set up with recognition and storage as CAD entities, it would serve as an excellent sketching tool.

4.3 SketchPad for Windows

It is clear from the discussion in the previous section that a sketcher in the form of an inking application that could represent and store design sketches as CAD entities would serve as an ideal sketching environment. With this design specification in view, *SketchPad for Windows* application was developed so that the designer could express the design strokes as sketches and the recognition algorithm would take care of recognizing the CAD entity in the sketch topology.

An example describing the working of the *SketchPad for Windows* application in a typical design environment is as described below.

Upon double-clicking the *SketchPad for Windows* icon [Figure 4.4], the application starts up and presents the application's sketch capture environment [Figure 4.5] to the user.

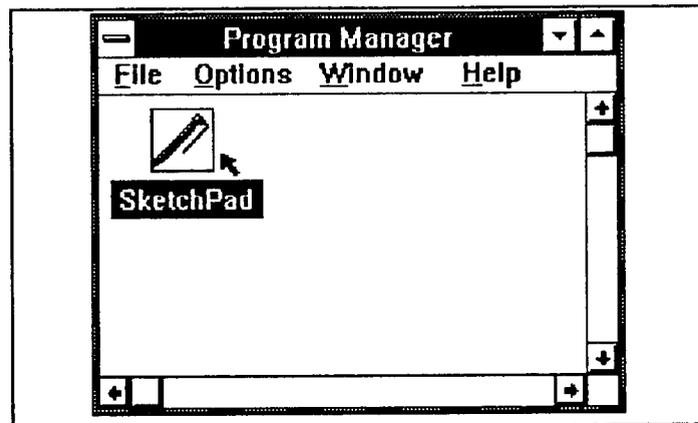


Figure 4.4: SketchPad for Windows Icon

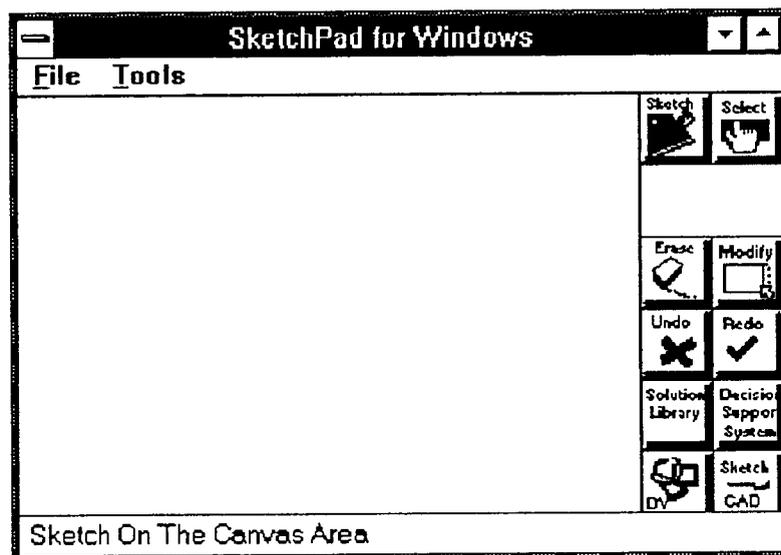


Figure 4.5: SketchPad for Windows

As can be seen from Figure [4.5], *SketchPad for Windows* supports a minimum of tool buttons for sketching and editing. The primary focus of the graphical user

interface is to offer sketching tools transparently, and not as hierarchial menus.

Now let us suppose that a designer intends to design a base plate. All that the designer has as information is the fact that it needs to be mounted with four bolts [Figure 4.6]. A mental image of the base plate is first developed in the designer's mind consisting of a rectangle defining the plate and the circles defining the holes.

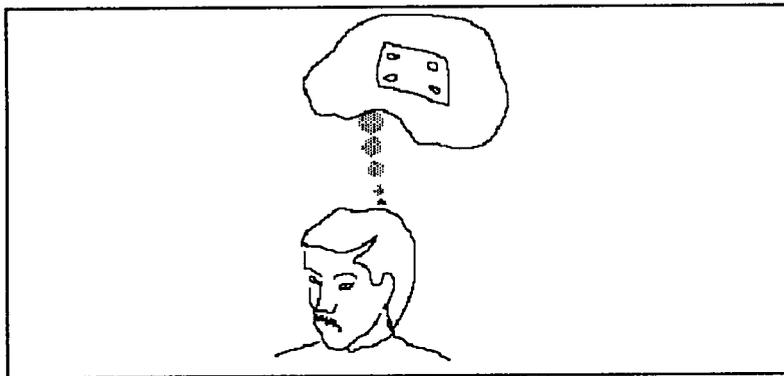


Figure 4.6: Design information in the mind

As the designer moves the cursor over the *Sketch* button, a message requesting the designer to *sketch on the canvas* is then displayed in the message area [Figure 4.7].

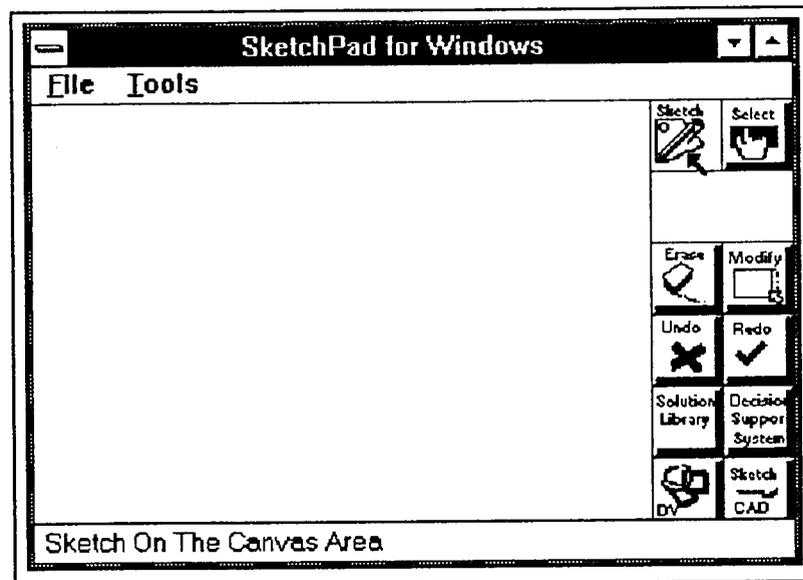


Figure 4.7: User messages

The designer then clicks on the *Sketch* button with the left mouse button. Having clicked the sketch button, the designer starts to stroke on the screen to represent his thoughts. To do so the user presses the left mouse button down and starts moving the mouse to express his sketch. As the user moves the mouse, a trail of the path chosen by user is represented by highlighting the pixels along the path. Since the base plate has four sides, the designer strokes in a linear fashion to represent one edge of the base plate [Figure 4.8].

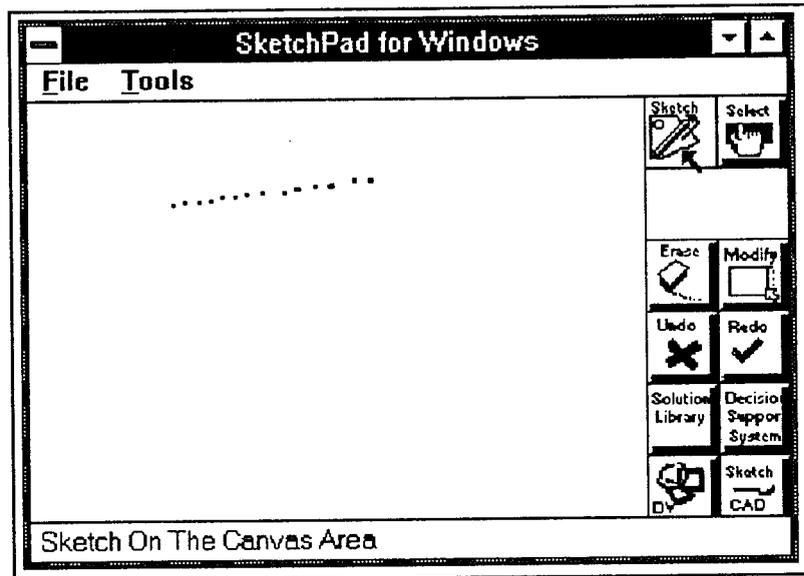


Figure 4.8: Linear sketch trail

Having expressed the thought as a stroke on the canvas, the user lets go the left mouse button. The recognition routine (discussed in section 4.4b) goes through the array of points stored and replaces the pixel trail with the recognized CAD entity. Since the user sketched linearly, recognition routine replaced the sketch stroke with a line entity [Figure 4.9].

It can be seen that the designer's intention was to draw a straight line and all that he/she did was to stroke in the drawing area in a linear fashion and then allowed the system to recognize the design intent. There are no drawing entity (line, circle, arc) icons or bitmaps forcing the designer to think in terms of those entities while representing the drawing. Having sketched one side of the base plate, the other three sides of the base plate are also sketched similarly to generate the sketch shown in Figure [4.10].

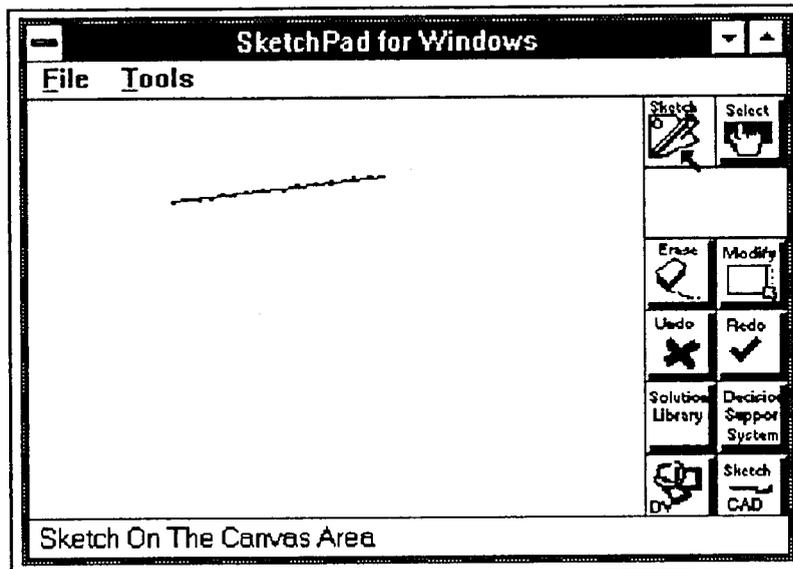


Figure 4.9: Line recognition

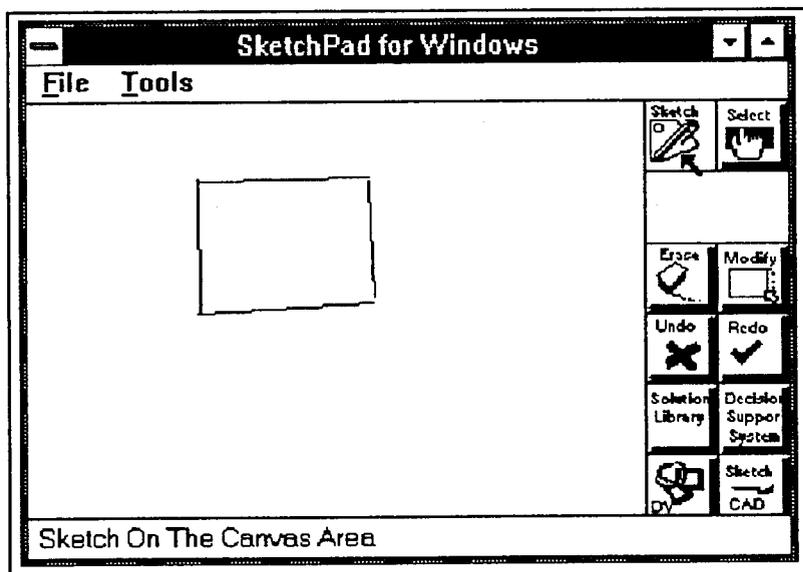


Figure 4.10: Base plate-Stage1

The user then strokes in a circular fashion [Figure 4.11] to sketch the bolt holes.

The entity recognition algorithm built into the application recognizes the stroke of the

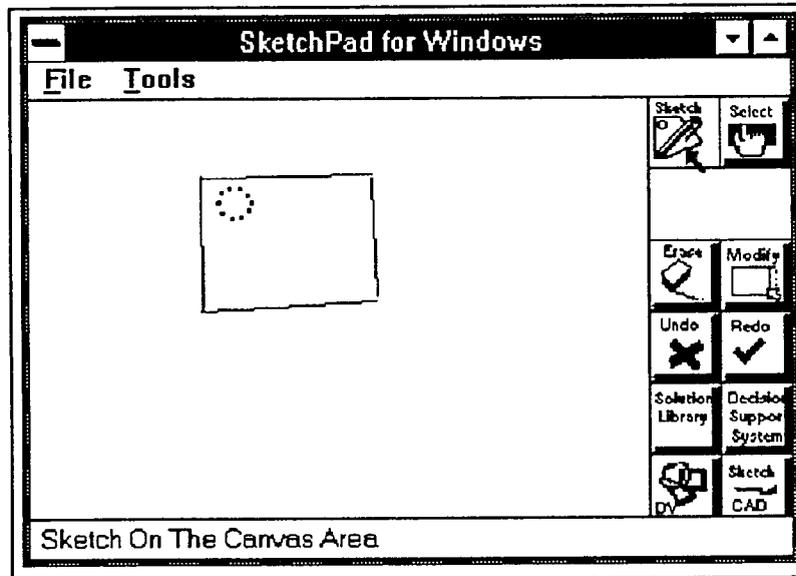


Figure 4.11: Bolt hole sketching

designer and replaces the stroke with a circle [Figure 4.12]. Since most engineering holes are circular in nature, the algorithm considers all holes as circular.

Since the entity-recognition algorithm uses approximation techniques, the sketches need to be refined. For example, the lines representing the boundary of the base plate need to be refined to represent the perpendicularity of the lines. In other words, the individual line entities need to be modified to represent the design intent.

The designer then clicks on the *select* button and clicks on the entity to be modified. The selected entity is automatically highlighted [Figure 4.13].

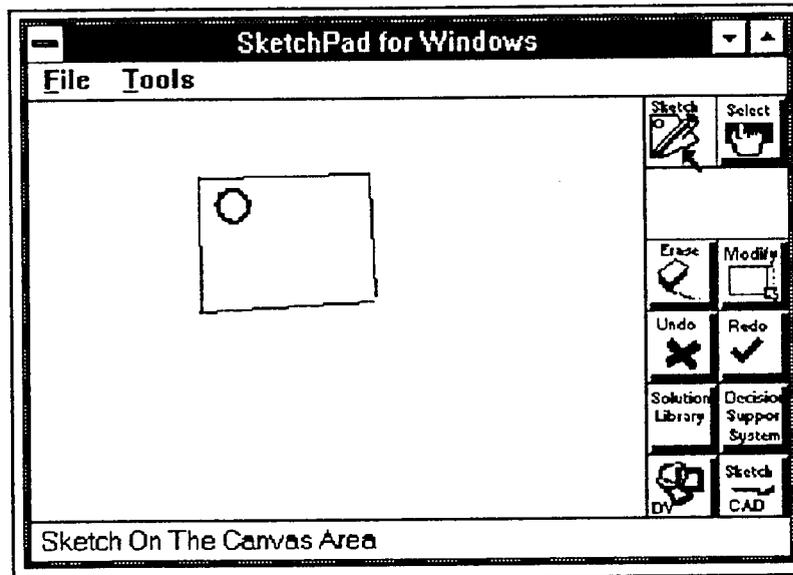


Figure 4.12: Circle recognition

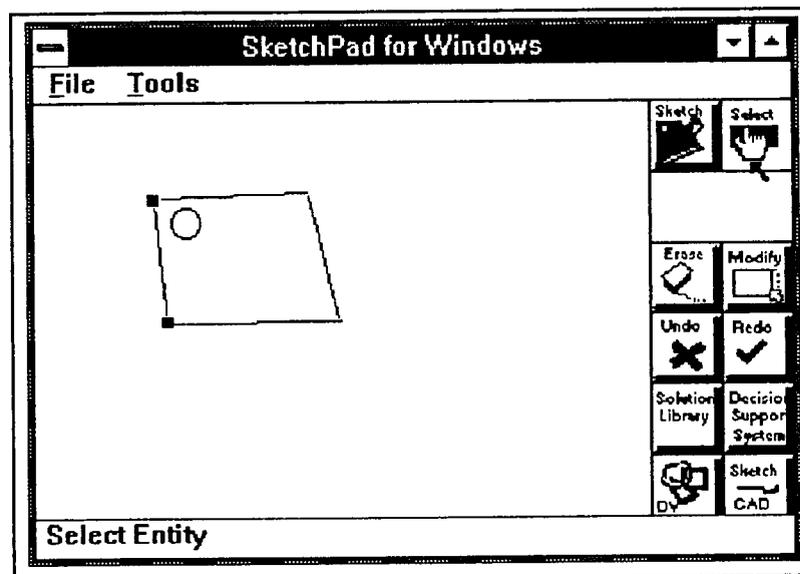


Figure 4.13: Entity selection

Since the selected entity needs to be modified, the designer clicks on the *modify*

button. The designer then clicks at that end-point of the entity that requires modification and moves the mouse (keeping the left mouse button pressed) to rotate (and resize) the entity about the other end point [Figure 4.14]. It is worth mentioning here that only the entity selected undergoes modification. The other line entities are also edited in a similar manner to obtain a refined sketch.

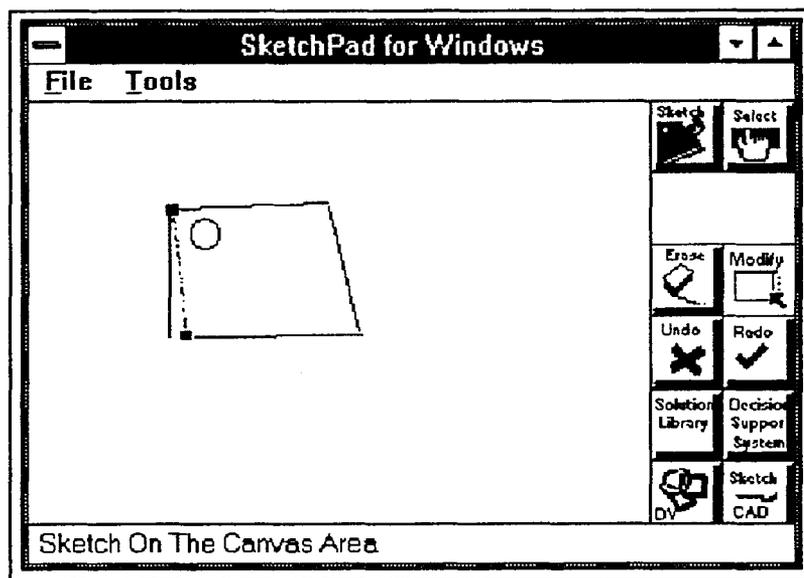


Figure 4.14: Entity modified

Since the designs in the conceptualization stage are not definite in terms of entity shape, size and orientation, the modification routine is very helpful as it allows the designer to modify the entity size and orientation.

The other three bolt holes represented as circles are also sketched to get the final sketch. Since this sketch was developed as a solution to a particular design problem, the designer wishes to store the design. He selects the *save* menu under the *File* menu and saves the file. Since the part was designed to perform a function, a record

of the functionality of the design would be desirable as a reference for other designers facing similar design problems.

Design specifications change often. For example, upon discussing with other design engineers, the designer realizes that the straight edge on the right hand side of the base plate is not ideal and that a curved surface needs to be used. The designer then selects the entity (as mentioned earlier) and clicks on the *erase* button and confirms the deletion with the click of the left mouse button. Having erased the entity, the designer proceeds to sketch the new surface. Clicking on the *sketch* button, once again, the designer strokes in a curvilinear fashion [Figure 4.15]. Upon releasing the left mouse button, the stroke is replaced with a circular arc [Figure 4.16].

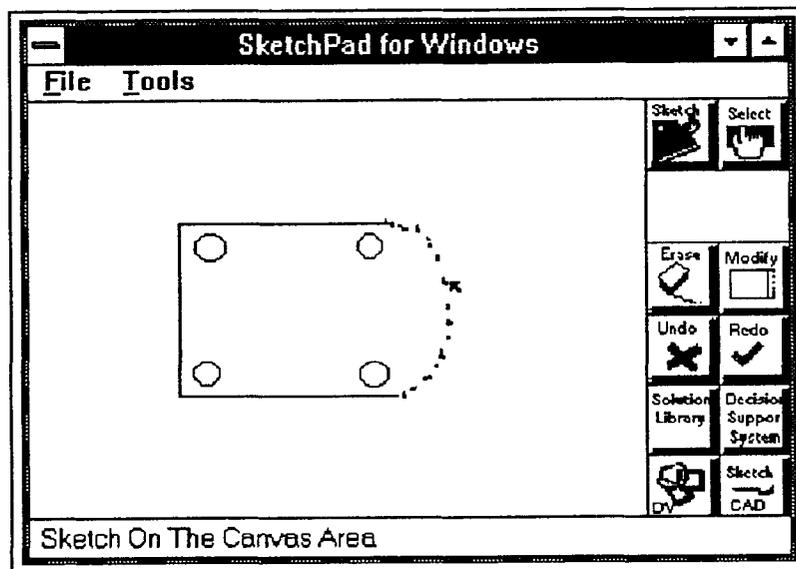


Figure 4.15: Arc trail

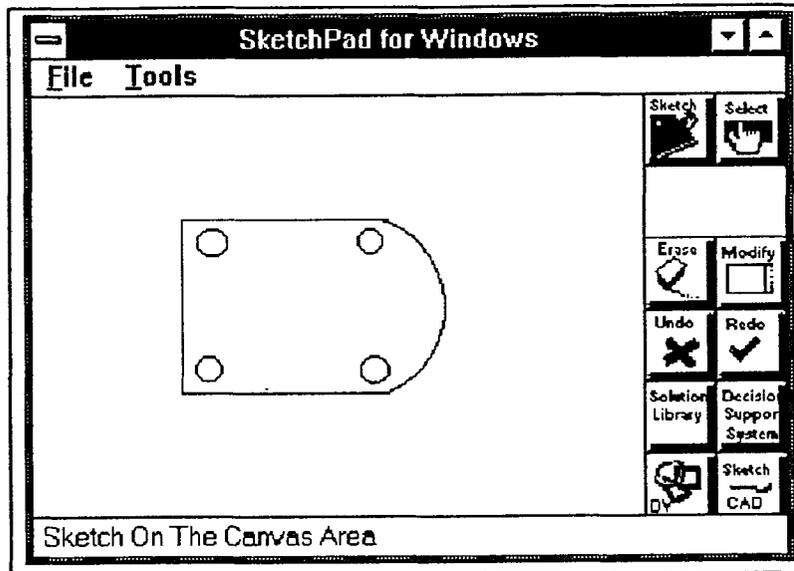


Figure 4.16: New design

In *SketchPad for Windows* an operation can be "undone" with an *undo* button at any stage in the sketching process . The *redo* button allows the designer to "undo" an "undo operation".

As mentioned earlier (see chapter 1), no design has commercial value unless the object designed can be made in a manufacturing shop. Manufacturing requires a detailed design. Although the sketch in *SketchPad for Windows* is very well defined at this stage, it lacks detailed dimensioning. Since CAD software are ideal for the detailed dimensioning, a tool that can convert the stored design sketches into CAD data is desirable. Although an interface to any CAD software could be generated, communicating with parametric CAD software such as DesignView allows refinement of the sketch. DesignView is ideally suited to address a wide range of design problems that engineers and designers encounter during preliminary design at the product, assembly, and component level, including:

- * Parametric Drawing
- * Piece Part Design
- * Geometric Studies
- * Kinematics
- * Tolerance Analysis and Fit-up
- * Mass Properties
- * Assembly Modeling

DesignView is based on the implementation of the *dimension-driven variational geometry* technology [*DesignView for Windows-User's Manual*] wherein a change in the dimension reshapes the geometry without having to recalculate and redraw everything affected by the changes. In DesignView geometry can also be constrained with equations. Convinced of the utility in converting data into DesignView format, a software routine that could convert sketch data into DesignView CAD data was developed.

In order to convert the sketch data the designer clicks on the *Sketch->CAD* button which converts the sketch data into DesignView CAD data. The convertor writes an output file "dv.dvx". The designer then clicks on the DesignView button to start DesignView.

After the DesignView application opens, the designer loads the converted drawing (dv.dvx) with the help of the .DVX macro in the *import* option under the *File* menu [Figure 4.17].

Although the sketch data is converted into a DesignView drawing, the latter may need modification. The designer therefore corrects the drawing using the editing features of DesignView. Having edited the drawing, the designer proceeds to dimension the drawing.

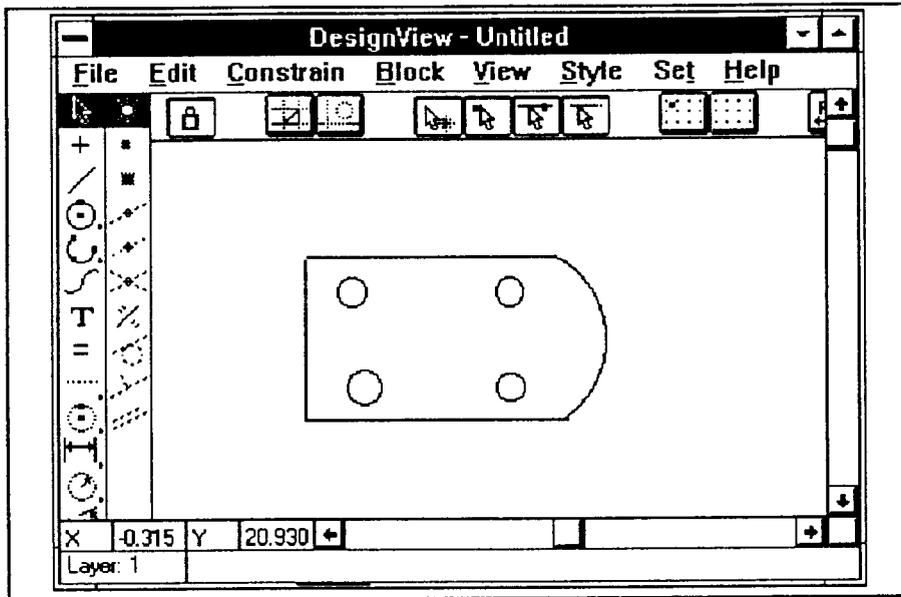


Figure 4.17: Importing the sketch into DesignView

Upon dimensioning the drawing, DesignView draws the dimension graphics and displays the current value of the dimension [Figure 4.18]. The designer then specifies the exact dimensions which causes DesignView to redraw the drawing while still maintaining all geometric constraints.

In order to represent the design relationships the designer can develop equations which are based on DesignView's *dimension-driven variational geometry* technology. Thus, any change in the equation causes DesignView to solve the equations, dimensions, and geometry constraints and redraw the geometry [Figure 4.19].

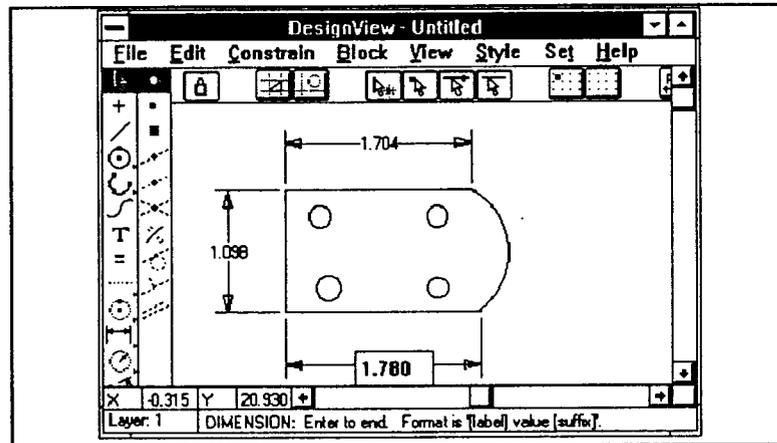


Figure 4.18: Dimensioning the drawing

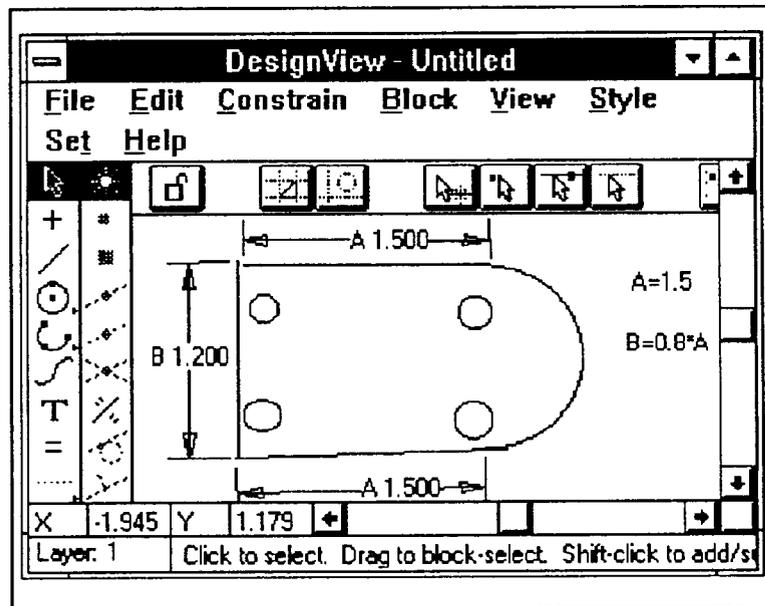


Figure 4.19: Dimension driven geometry

Although this example refers to utilizing the *dimension-driven variational geometry* technology of DesignView, the designer can also solve for parameters. It is thus clear from the above demonstration that *SketchPad for Windows* is an ideal tool for capturing sketches from the conceptual design stage into the detail dimensioning stage.

The Solution Library [Wood 95] allows the designer to search for design solutions with the help of function searches. In order to capture the design sketch the designer presses the right mouse button down in the drawing area and moves the mouse (keeping the right mouse button pressed) so as to enclose the sketch with a selection-rectangle [Figure 4.20]. After having enclosed the desired area with the selection-rectangle, the designer releases the right mouse button. The sketch area enclosed by the selection-rectangle is now captured as bitmap and is temporarily stored in the Microsoft Windows ClipboardViewer.

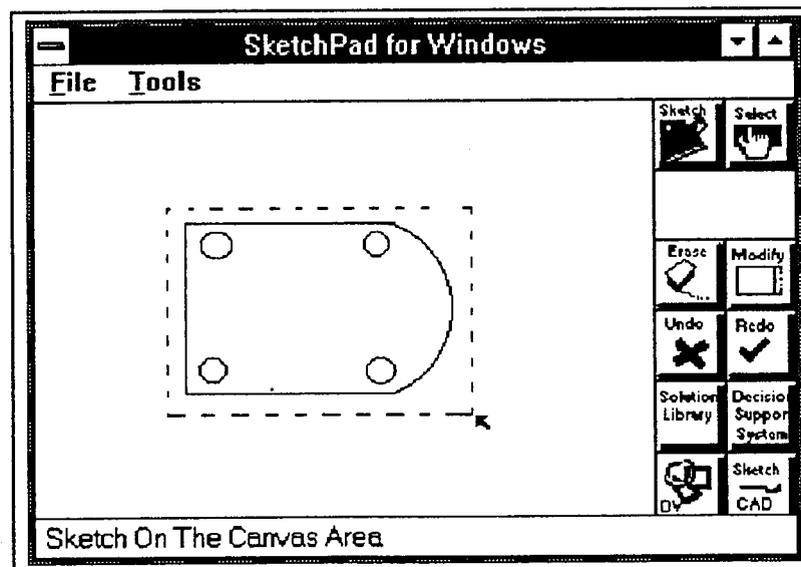


Figure 4.20: Capturing sketch information

The designer then clicks on the *Solution Library* button causing the *Solution Library* application to open. Opening the feature definition form, the designer pastes the captured bitmap in the image area. The designer then defines the function verbs for the design [Figure 4.21] and saves the information as a new entry in the *Solution Library*. The design and its functionality is now recorded in the *Solution Library* for use in other designs.

The screenshot shows a software interface titled "Feature-Definition". It contains several input fields and a drawing area. The "Feature Name" field contains "base plate". The "Goal-Function Primary" field contains "hold". There are seven "Secondary" function fields: "Secondary 1" contains "position", "Secondary 2" is redacted with a black box, and "Secondary 3" through "Secondary 7" are empty. To the right, the "Feature ID" field is empty. Below the text fields is a "Feature Shape" area containing a technical drawing of a rectangular plate with four circular holes and a rounded right edge. At the bottom of the window, there is a toolbar with buttons for "Edit", "End", "Exit", and a set of navigation arrows (left, right, and double arrows).

Figure 4.21: Design capture in Solution Library

This example assumes that the designer is independently working on the design. However, if a team of design engineers are designing the same, a tool that analyzes the design decisions made by the designers would be beneficial for design justification. The Decision Support system [Herling *et.al.* 95], based on OREO [Ullman 93] theory, is a tool aimed at comparing and analyzing alternative designs.

The Design Support system accepts bitmaps also as design information. Therefore, the captured bitmap from *SketchPad for Windows* can be pasted into this application just as it was done for the Solution Library.

4.4 Feature implementation

All the features of *SketchPad for Windows*, referred to in the demonstration above, have been implemented with Object Windows Library (OWL 2.0), the object-oriented Microsoft Windows application development tool-kit from Borland.

4.41 Implementing SketchPad for Microsoft Windows

The graphical user interface of *SketchPad for Windows* has been implemented using the window objects of OWL. Window objects comprise of the windows, menus, toolboxes, message bars *etc.*, and are as shown in Figure [4.22].

The main window (TMainWnd) has been derived from the TDecoratedFrame class, while the client window (TCanvas) has been derived from the TWindow class. The menu, toolbox and message bar for the application is assigned to the main window while the client window handles all the drawing utilities.

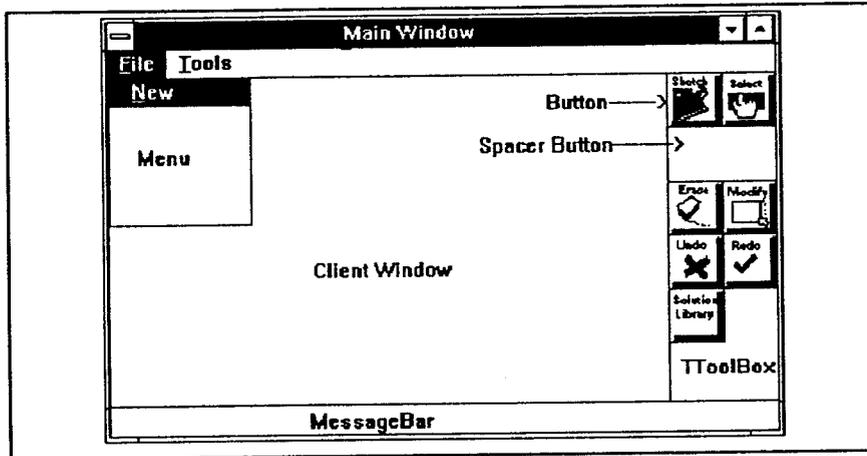


Figure 4.22: Window objects in SketchPad for Windows

The implementation of these window objects as classes in *SketchPad for Windows* is as shown in Figure [4.23].

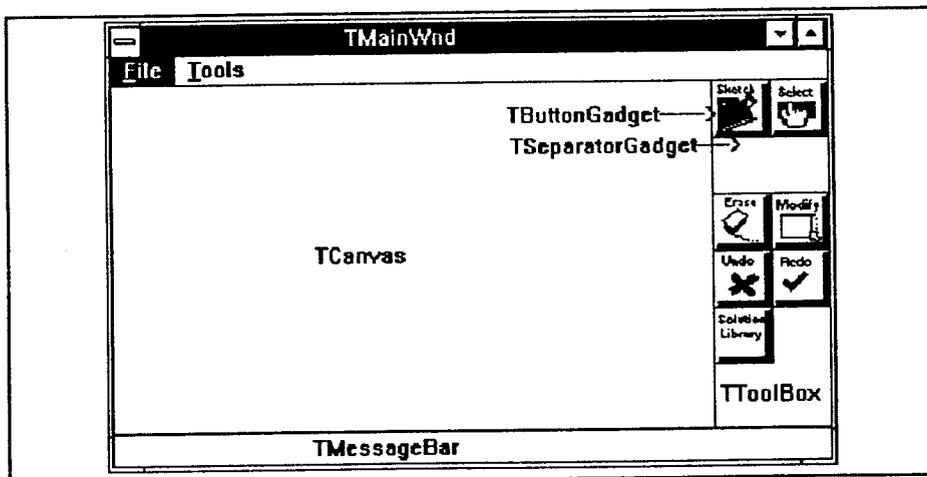


Figure 4.23: Window classes in SketchPad for Windows

Upon double-clicking the *SketchPad for Windows* application icon, *OwlMain* starts the sketching application. *OwlMain* is the Object Windows Library (OWL) implementation of the standard *main* function of C and C++ programming language and is implemented in *SketchPad for Windows* as shown below.

```
int OwlMain (int, char *[] ){
    return TDCSApp( ).Run( );
}
```

Application programs written with OWL need to support at least two kinds of objects, an application object and a window object. **TDCSApp**, as shown below, is the application object that manages all the OWL objects in the program and sets the application's message loop running in *OwlMain*.

```
class TDCSApp : public TApplication {
public:
    TDCSApp():TApplication(){
        void InitMainWindow();
    };

    void TDCSApp :: InitMainWindow ( ) {
        //Construct the client "Canvas" window
        TWindow *client = new TCanvas(0,0);
        //Construct the main window "frame"
        TDecoratedFrame *frame = new TMainWnd(0,"DCS for
Windows",client,TRUE);
        //Set MainWindow to frame
        SetMainWindow(frame);}
};
```

As can be seen from the code above, the *InitMainWindow* member function of **TDCSApp** instantiates the window objects **TMainWnd** and **TCanvas**.

The `TMainWnd` class, which is the main window class for *SketchPad for Windows*, has been publicly derived from the `TDecoratedFrame` class of Object Windows Library as shown below.

```
class TMainWnd : public TDecoratedFrame{
protected:
    TToolBox *toolbox;
public:
    TMainWnd(TWindow *parent, const char far *title,
            TWindow *client, BOOL trackMenuSelection);
protected:
//Member Functions are defined below
//"File" menu functions
void CmFileNew();           //Open a new file
void CmFileOpen();         // Open an existing file
void CmFileSave();         // Save the current file
void CmFileSaveAs();       // Save the file as...
void CmExit();             //Exit the application
// Toolbox button functions
void CmSketch(WPARAM Id); //Sketch on the screen
void CmSelect(WPARAM Id); //Select entity on screen
void CmErase(WPARAM Id); //Erase the entity selected
void CmUndo(WPARAM Id);   //Undo the current operation
void CmRedo(WPARAM Id);   //Redo the current operation
void CmModify(WPARAM Id); //Modify the entity size
DECLARE_RESPONSE_TABLE(TMainWnd);
};
```

Borland C++ 4.0 version supports the `RESPONSE_TABLE` feature so that each member function can be associated with an activation command message or command identity (commonly called id). The `RESPONSE_TABLE` declared for the `TMainWnd` class defines the functionality of the `TMainWnd` file and tool member functions as shown below.

```

DEFINE_RESPONSE_TABLE1(TMainWnd,TDecoratedFrame)
    EV_COMMAND(CM_FILENEW,CmFileNew),
    EV_COMMAND(CM_FILEOPEN,CmFileOpen),
    EV_COMMAND(CM_FILESAVE,CmFileSave),
    EV_COMMAND(CM_FILESAVEAS,CmFileSaveAs),
    EV_COMMAND(CM_EXIT, CmExit),
    EV_COMMAND_AND_ID(CM_SKETCH,CmSketch),
    EV_COMMAND_AND_ID(CM_SELECT,CmSelect),
    EV_COMMAND_AND_ID(CM_ERASE,CmErase),
    EV_COMMAND_AND_ID(CM_UNDO,CmUndo),
    EV_COMMAND_AND_ID(CM_REDO,CmRedo),
    EV_COMMAND_AND_ID(CM_MODIFY,CmModify),
END_RESPONSE_TABLE;

```

For example, clicking on the "New" menu-item in the File menu causes the Microsoft Windows CM_FILENEW command-message to be generated. The RESPONSE_TABLE defines the member function to be associated with this command message by the EV_COMMAND (CM_FILENEW, CmFileNew) statement. Similarly the EV_COMMAND_AND_ID statement defines the member function to be associated in response to a generated *command id*. For example, when the user clicks on any of the bitmapped buttons, a specific *command id* is generated. The *command id* to be generated when the bitmapped button is selected is defined in the TMainWnd constructor.

The menu for the application is assigned in the TMainWnd constructor. The TMainWnd class constructor, as shown in the code below, also handles the instantiation of the TToolBox, TButtonGadget, TSeparatorGadget, and TMessageBar classes.

```

TMainWnd ::TMainWnd(TWindow *parent, const char far *title,
TWindow *client, BOOL trackMenuSelection):
TDecoratedFrame(parent,title,client,trackMenuSelection){
    //Create a pointer to the TButtonGadget object
    TButtonGadget *bgadget;
    //Create a pointer to the TSeparatorGadget object
    TSeparatorGadget *sgadget;
    //Assign the menu to TMainWnd
    AssignMenu(WINDCS);
    //Instantiate the TToolBox class
    toolbox = new TToolBox(this);
    //Set the hint mode for the bitmapped buttons
    toolbox->SetHintMode(TGadgetWindow::EnterHints);
    //Instantiate the TButtonGadget object and associate command id
    bgadget = new TButtonGadget(BMP_SKETCH,CM_SKETCH);
    //Insert the instantiated button gadget in the toolbox object
    toolbox->Insert(*bgadget);
    bgadget = new TButtonGadget(BMP_SELECT,CM_SELECT);
    toolbox->Insert(*bgadget);
    //Instantiate the TSeparatorGadget
    sgadget = new TSeparatorGadget(10);
    //Insert the instantiated separator gadget in the toolbox object
    toolbox->Insert(*sgadget);
    sgadget = new TSeparatorGadget(10);
    toolbox->Insert(*sgadget);
    bgadget = new TButtonGadget(BMP_ERASE,CM_ERASE);
    toolbox->Insert(*bgadget);
    bgadget = new TButtonGadget(BMP_MODIFY,CM_MODIFY);
    toolbox->Insert(*bgadget);
    bgadget = new TButtonGadget(BMP_UNDO,CM_UNDO);
    toolbox->Insert(*bgadget);
    bgadget = new TButtonGadget(BMP_REDO,CM_REDO);
    toolbox->Insert(*bgadget);
    Insert(*toolbox,TDecoratedFrame::Right);
    //Instantiate the TMessageBar class
    TMessageBar *mesgbar = new TMessageBar(this);
    mesgbar->SetText("Select An Option");
    //Insert the messagebar at the bottom in the TMainWnd frame
    Insert(*mesgbar,TDecoratedFrame::Bottom);
}

```

The menu, `WINDCS`, in order to be displayed should be included in the `.RC` file that is linked and bound with the executable of the `.CPP` file (`.RC` files define the resources, `.CPP` files deal with Object Windows Library code and the `.DEF` file defines the Windows definition file). The implemented menu supports tracking so that when the user moves the mouse over the different menu items, it generates the respective user messages in the message area.

The `TButtonGadget` is instantiated with the appropriate bitmap file and associated with the respective *command id*. The instantiated `TButtonGadgets` and `TSeparatorGadgets` are inserted into the `TToolBox` object, which is instantiated in the `TMainWnd` object. `TMessageBar`, the object that prompts user messages, is instantiated and inserted at the bottom of the `TMainWnd` object.

`TCanvas`, the drawing area or canvas (as it is called in the graphics industry) is publicly derived from the OWL `TWindow` class as shown below. `TWindow` provides window-specific behavior and encapsulates many Windows API functions.

```
class TCanvas : public TWindow {
protected:
    //Boolean parameters to check state of mouse button, //drawing and
    //modification button selected.
    BOOL lbuttondown,rbuttondown;
    BOOL Sketch,Select,Erase,Undo,Redo,Modify;
    //Pointer to a TPen object
    TPen *pen;
    //SketchDC is the Device Context for Sketching
    //CaptureDC is the Device Context Bitmap Capturing
    TClientDC *SketchDC, *CaptureDC;
    //CopyRect is the rectangular area copied as a bitmap
    TRect CopyRect;
    //Pointers to TObjectList, TObject, TListNode objects
    TObjectList *List;
    TObject      *RedoObject;
};
```

```

//Point, number definitions for sketch recognition
TPoint Begin, End, AppxBgn, AppEnd;
int Num_Of_Points;
//Statistical variables
long double sigx,sigy,sigxy,sigx2,sigy2,Xpt[1000],Ypt[1000];
long double meany,totalvar,unexpvar,minx,miny, maxx,maxy;
double det,radius,value; //Determinant and radius variables
//Slope,intercept and coeff of correlation for line
float a,b,coeff_of_corr;
FILE *fp;
public:
TCanvas (TWindow *parent, const char far *title);
~TCanvas( ){ delete pen; delete SketchDC; delete CaptureDC; delete List;
delete RedoObject; delete HiliteNode;}
protected:
void EvLButtonDown(UINT, TPoint &point);
void EvLButtonUp(UINT, TPoint &point);
void EvLButtonDbIClk(UINT, TPoint &point);
void EvMouseMove(UINT, TPoint &point);
void EvRButtonDown(UINT, TPoint &point);
void EvRButtonUp(UINT, TPoint &point);
void Paint(TDC &PaintDC, BOOL, TRect&);
LRESULT PmStatus(WPARAM number, LPARAM);
DECLARE_RESPONSE_TABLE(TCanvas);
};

```

As can be seen from the definition of the TCanvas class, the class supports data and member functions that aid drawing and editing functions. The member functions, performing drawing and editing tasks, have been declared using the RESPONSE_TABLE. Since each mouse button performs various operations in the application, boolean parameters are used to distinguish the operations. The data members of TCanvas class consists of statistical variables which are used for entity recognition and pointers to the link list objects (TObjectList, TObject and TListNode).

Sketches on the window need to be stored. Commonly used storage data types are arrays and link lists. Since arrays require allocation of memory before running a

program, a link list that allocates memory as and when needed, has been used to store the drawing data. Unlike arrays, link lists do not allocate memory contiguously and can be scattered everywhere. Therefore in order to maintain connectivity of data a pointer to the next data is specified [Figure 4.24].

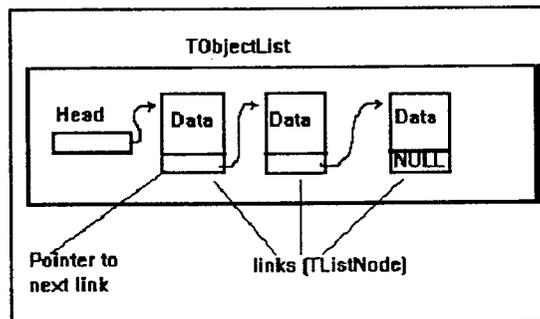


Figure 4.24: Link list

TListNode, the object representing the individual links in the link-list, stores the drawing object data and a pointer to the next object data as shown in the code below.

```

class TListNode{
    friend class TObjectList;
    friend class TObjectListIterator;
    public:
        TListNode *Next; //Pointer to next object
        TObject *ListObject;//Current object
    public:        //Constructor
        TListNode(TObject *Obj,TListNode *N=0){
            ListObject=Obj;
            Next      =N;
        }
        void Draw(TDC &DevCon){//Member functions
            ListObject->Draw(DevCon);//to draw when Wind
        }
        void HiLite(TDC &DevCon) {
            ListObject->HiLite(DevCon);
        }
};

```

TObjectList, the link list of objects, performs the operations of adding, inserting, traversing and deleting the list nodes and is shown in the code below.

```

class TObjectList {
    friend class TObjectListIterator;
protected:
    TListNode *Head;
    virtual TListNode* DoInsert(TObject* Obj, TListNode* N)
        { return new TListNode(Obj, N); }
    virtual TListNode* DoAppend(TObject* Obj, TListNode* N);
    virtual TListNode* DoDelete(TObject* Obj, TListNode* N);
public:
    TObjectList(){ Head=0;}
    ~TObjectList();
    void Insert(TObject *Obj) { Head = DoInsert(Obj, Head); }
    void Append(TObject *Obj) { Head = DoAppend(Obj, Head); }
    void Del(TObject *Obj) { Head = DoDelete(Obj, Head); }
    void Traverse(TDC &DevCon);
    void BoundingRect(TPoint TopLeft, TPoint BottomRight);
    TListNode* SelectedObject(TPoint P);
    TListNode* UndoNode();
    int GetObjType();
};

```

Once the list is created, it's easy to step through all the members, displaying them (or performing other operations). The iterator class, **TObjectListIterator**, performs this function and is defined as shown below.

```

class TObjectListIterator{
    TListNode* Objptr;
public:
    TObjectListIterator(TObjectList& List) {
        Objptr = List.Head;
    }
    TObject* operator()() {
        TObject* Obj = Objptr ? (Objptr->ListObject) : 0;
        Objptr = Objptr ?
        Objptr->Next : 0;
        return Obj; }
};

```

TObjectList stores the sketch information on the screen in the form of drawing objects. Drawing objects in the application consist of lines, circles and arcs. These drawing objects are implemented as **TLine**, **TCircle** and **TArc** classes respectively. These classes have been derived publicly from the parent **TObject** class as shown in Figure [4.25].

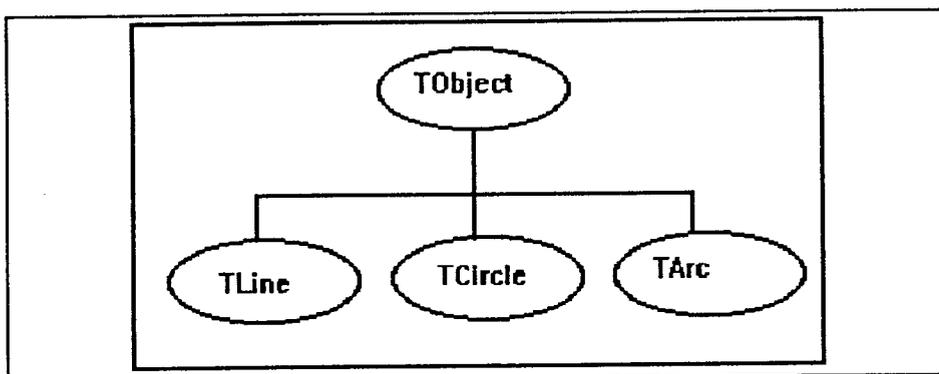


Figure 4.25: Class hierarchy

TObject, the parent drawing object class shown below, stores the object type and its bounding rectangle, calculates the object's distance from a point, checks whether a point is within an object's boundary, highlights a selected object and draws the object whenever Microsoft Windows sends a WM_PAINT message.

```

class TObject {
private:
    //To store the object type (LINE,ARC,CIRCLE,ELLIPSE etc)
    int ObjType;
public:
    TObject(){} // No argument constructor
    TObject(int Type){ ObjType=Type;} //Single argument constructor
    //Member Functions
    virtual int GetObjType() { return ObjType;}
    virtual void SetObjType( int Type){ ObjType=Type;}
    //Stores the bounding rectangle
    virtual void BoundingRect(TPoint, TPoint){}
    //Check distance from point clicked to the center of the entity
    virtual double Distance(TPoint){}
    //Checks whether the clicked point lies in bounding rectangle
    virtual BOOL Contains(TPoint){}
    //Hilites when an entity is selected
    virtual void HiLite(TDC &DevCon){}
    //Modify the size and shape of entity
    virtual void Modify(TPoint){}
    //Draws the entity
    virtual void Draw(TDC &DevCon){}
};

```

The member functions of the derived (TLine, TCircle and TArc) classes over-ride the virtual functions of the base TObject class. As an example, the *virtual void Draw(TDC &DevCon)* member function of the TLine class is used to repaint the window with a line object whenever Microsoft Windows sends a WM_PAINT message to the TCanvas object and is shown in the code below.

```
void TLine::Draw(TDC &DevCon){
    DevCon.MoveTo(FirstCorn);
    DevCon.LineTo(SecondCorn);
}
```

However, when the window needs to be repainted with a circle object, the *virtual void Draw(TDC &DevCon)* member function of TCircle class is called as shown in the code below.

```
void TCircle :: Draw(TDC &DevCon) {
    DevCon.Ellipse (FirstCorn, SecondCorn);
}
```

It is worth mentioning here that since the Windows API function does not support a *Circle* function, the *Ellipse* function defined by a bounding square is used. It can be seen from the above example that by deriving the TLine, TCircle and TArc class from the parent TObject class we are able to extend the functionality of the member functions derived for each of the derived classes.

4.42 Recognition Algorithm

SketchPad for Windows recognizes the design intent of the designer and replaces the mouse trail with the respective CAD entity. The recognition algorithm in SketchPad is largely based on the simple and popular *least-squares* curve fitting algorithm.

The sampled (x,y) points from the mouse are fed into the *least-squares* model. The relation between the y coordinates and x coordinates is then determined and expressed as *coefficient of determination* (Very often in regression analysis the dependent variable is not completely explained by the independent variable and there is explained and unexplained variation. The explained variation is called as

coefficient of determination). A *coefficient of determination* of 1 indicates that all of the variation in the independent variable is explained by the dependent variable. However, a coefficient of 0 indicates none of the variation in the independent variable is explained by the dependent variable.

The use of the *coefficient of determination* in recognizing the CAD entity can be represented as shown in the table below.

Coefficient of determination (r^2)	Meaning	Algorithm to use
$0.995 < (r^2) < 1$	Line	Linear regression
$(r^2) < 0.995$	Circle/Arc	Calculate included angle and center

Table 4.1: Recognition parameters

The sketch recognition process can be summarized as shown in the Figure [4.26]. The recognition value of 0.995 has been used to compensate for the slow sampling rate of the system.

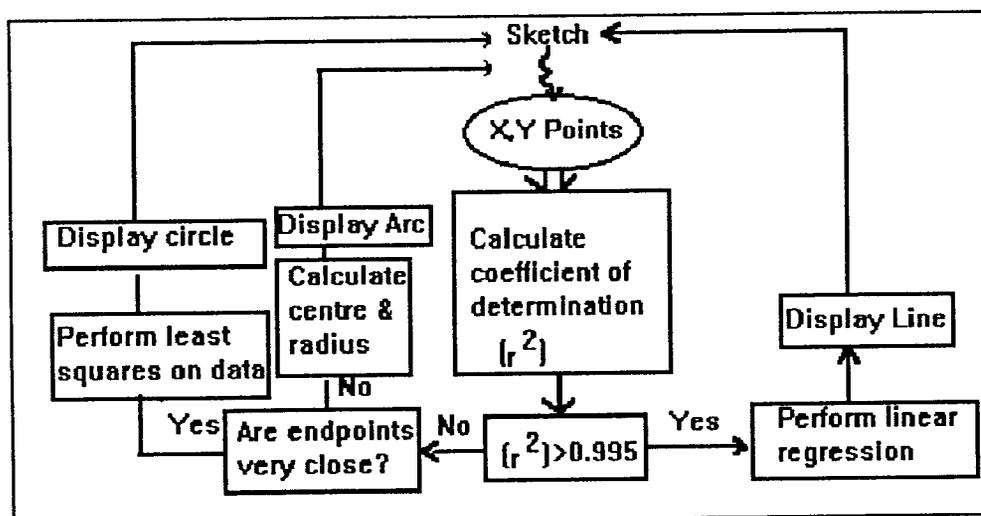


Figure 4.26: Sketch recognition algorithm

4.43 Communicating with other applications

As mentioned earlier in the chapter in the example, *SketchPad for Windows* communicates with other Microsoft Windows applications such as Solution Library, Decision Support system and DesignView.

SketchPad for Windows communicates with Solution Library and Decision Support system by allowing the user to copy some portions or all of the sketch data as a bitmap with the OWL Clipboard object. Implementing the clipboard function allows the user to copy bitmaps into any Microsoft Windows application supporting clipboard cut and paste functions.

The current version of *SketchPad for Windows* communicates with DesignView for Windows by converting the sketch data into the .DVX format of DesignView. The .DVX file format is determined by saving a DesignView file in that format and then

viewing its contents. The .DVX file consists of a set of setup commands, points information, information of the entity defined by those points, and a set of termination commands. Having analyzed the .DVX file format, a convertor was developed that scanned the sketch file data and wrote a .DVX file.

5. Conclusions

5.1 Summary

The *SketchPad for Windows* application is an ideal sketching tool to assist the design engineer in the design process by capturing *back-of-the-envelope* sketches.

Developed for the Microsoft Windows environment, *SketchPad for Windows* offers the following to the designer/engineer:

- 1) A *transparent* graphical user interface for sketching and editing

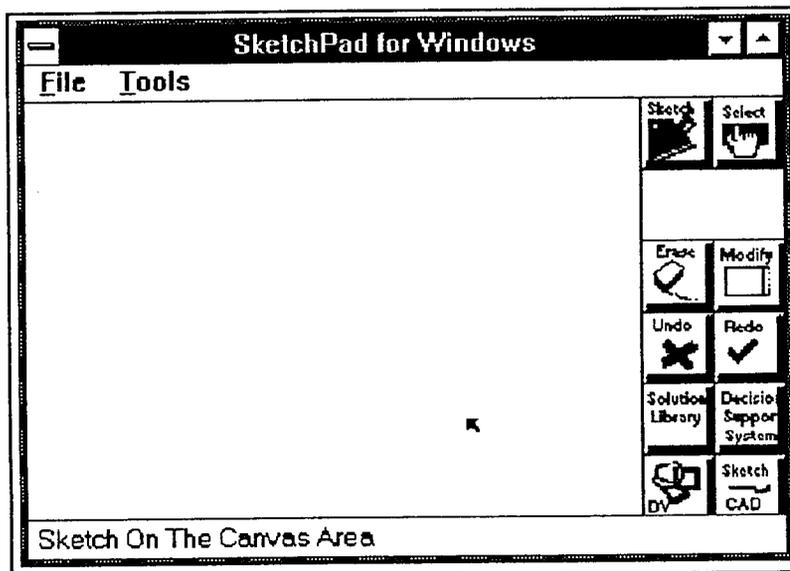


Figure 5.1: Transparent graphical user interface

With this graphical user interface [Figure 5.1], the design engineer no longer needs to wade through a hierarchy of menus as in the case of CAD software.

2) A sketching environment similar to the pencil and paper environment.

The user sketches as he would sketch on a piece of paper and the sketch recognition algorithm (described in section 4.42) recognizes what the user intended. The impressive feature of the recognition algorithm is the fact that sketches are corrected as they are drawn.

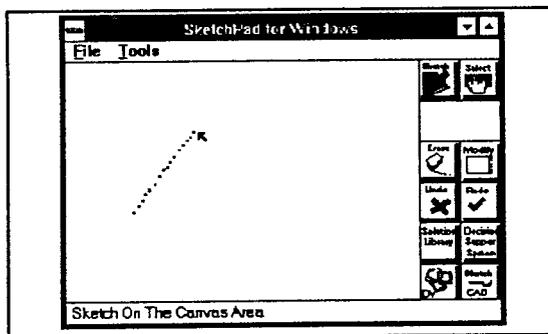


Figure 5.2: Linear trail

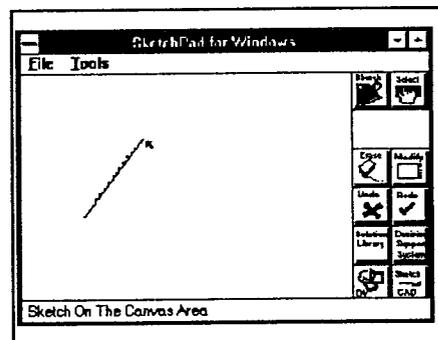


Figure 5.3: Line recognition

As can be see from the figures above, the user strokes [Figure 5.2] in a linear fashion keeping the left mouse button down and upon releasing the left mouse button, a line entity is displayed [Figure 5.3]. The recognition algorithm recognizes the design primitives (line, circles, arcs) topology from the trail of pixels. Therefore, the user no longer has to think in terms of representing the design topology as lines, circles, or arcs. The strokes that the user makes represent design decisions and the software stores them as lines, circles, and arcs.

3) A sketching tool that assists the designer in the design process

SketchPad for Windows is integrated with the *Solution Library* [Wood 95], *Decision*

Support system [Healing *et. al.* 94] and DesignView parametric CAD software. The user can sketch design solutions in the sketching application, look for other possible designs with the help of function searches in the Solution Library, store the design sketch as a design issue in the Decision Support system for further evaluation and perform detail and parametric designing with DesignView.

As an example, the user can sketch some designs on *SketchPad for Windows*. The user then decides to look for existing designs for the part functionality that he/she is looking for. In order to do so, the user uses the function based search of the *Solution Library*. Upon locating the required design, the user then modifies the current sketch based on the ideas from the retrieved design. After defining the sketch completely, the user copies the new sketch information with the clipboard feature and pastes it into the *Solution Library* as a new entry and adds the search functions to it. On the other hand if the user wanted to compare the design sketches made by a team of design engineers, he/she could use the *Decision Support* system for evaluation. If the user wished that his/her design idea be also recorded as an issue, the bitmap image of the sketch could be pasted into the Decision Support system. As can be seen from the explanation and from Figure [5.4], the integration of *SketchPad for Windows*, *Solution Library* and *Decision Support* system would comprise a design capture system; a tool ideal for the conceptualization stage.

Manufacturing requires that the design sketches be detailed. Therefore, the user can also transfer the sketch from *SketchPad for Windows* into DesignView CAD format using the convertor built into the application. Currently the convertor offers interface to DesignView in the form of DesignView Exchange Macro (.DVX format).

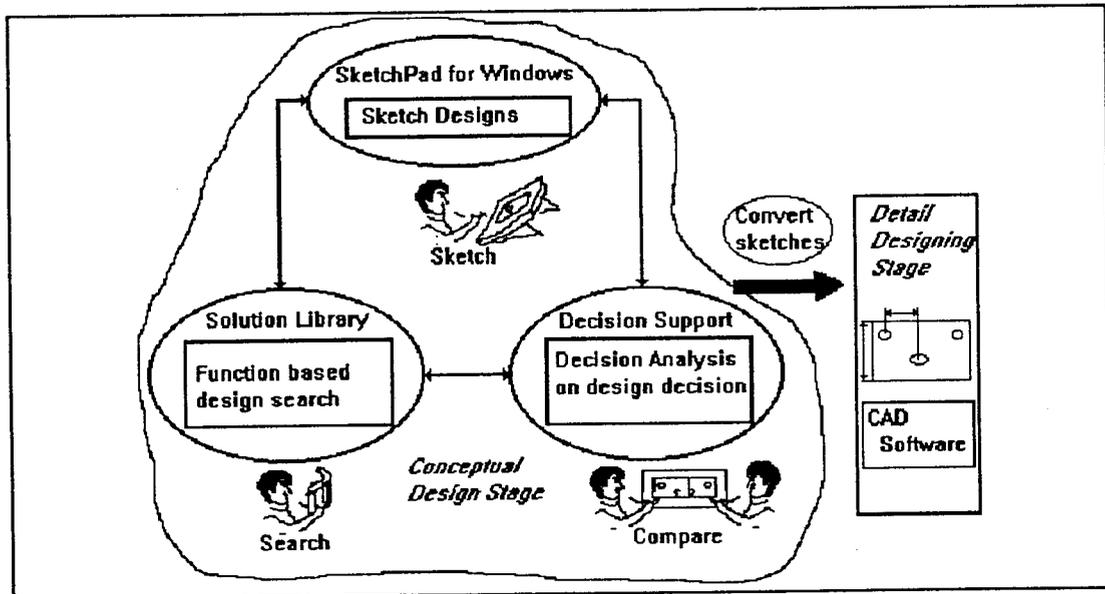


Figure 5.4: Design Capture System

5) Ability to save and recall sketches

The sketching applications by Fang [Fang 88] and Hwang [Hwang 91] did not allow the user to perform file operations on the sketch. An error occurring in the code meant that the code would crash and the sketch data would be lost. Valuable design sketches would thus be lost. Moreover the ability to perform file operations on the design sketches meant that the user could retrieve previous sketches and work on them.

Therefore, file opening and saving operations were implemented in *SketchPad for Windows*. Currently, the application can perform file operations on files stored in the *SketchPad for Windows* format.

4) A powerful sketching tool for the Microsoft Windows environment

The most popular hardware in the real world is the PC (personal computer). Currently most PC's support Microsoft Windows 3.1. Therefore, application vendors are developing more and more applications on the Windows platform. Even UNIX based applications are being re-written for the Microsoft Windows environment. Although the current version of *SketchPad for Windows* does not support DDE (Dynamic Data Exchange), the code can be enhanced to support DDE.

SketchPad for Windows, as mentioned earlier, has been written in C++ in the form of modules. *SketchPad for Windows* offers the following to the application developers intending on extending the features implemented in this code:

1) A modular program

The code for *SketchPad for Windows*, has been written in Borland C++ using the principles of object-oriented programming. The utility (see chapter 3.1) of object-oriented programming lies in the fact that the program is organized allowing non-programmers to understand and comment on the code. The modular programming approach incorporated makes room for re-use of the application code in future work with principles of inheritance. The modular nature of the code allows modification of the portions of the code without affecting the rest of the code.

This sketching application has been targeted for the 2D sketching environment. The object-oriented nature of the code allows the program to be extended to a 3D environment also.

2) A sketching application that is not restricted to a particular system configuration.

SketchPad for Windows runs on any hardware that supports Microsoft Windows 3.1 and higher versions of Microsoft Windows. The application developer no longer has

to worry about the device drivers as Microsoft Windows supports a host of device drivers.

5.2 Limitations of SketchPad for Windows.

SketchPad for Windows is an application that mimics the pencil and paper environment. The user sketches as he would sketch on a piece of paper and the recognition algorithm within the application recognizes the entity topology and replaces the pixel trail with an appropriate CAD entity. The recognition algorithm, as explained earlier, is based on the simple linear regression analysis model. The arc algorithm is dependent on the sampling of point for the stroke. If the points are not sampled properly, the recognition algorithm fails.

The current version of *SketchPad for Windows* works on IBM systems supporting Microsoft Windows 3.1 with the mouse as the locator device. This is not an ideal operating environment as it does not really mimic the pencil and paper environment. With the mouse and the screen, strokes are not controlled and the user does not get the feel of the drawing paper as he/she would do when drawing on a piece of paper. The ideal operating environment is the pen based hand-held computer from NCR (refer to the section on recommendations for future research).

The current version of *SketchPad for Windows* does not support:

- * Printing of sketches.
- * Capture of geometric (tangency, perpendicularity *etc.*) and spatial (whether one entity is within another *etc.*) intent.
- * Zoom and pan features. Since sketches need to be refined before being converted into CAD systems, it is desirable to have zoom and pan features for entity modification.
- * Entering textual information.

5.3 Recommendations for future research

The primary aim of this thesis was to develop an intelligent and interactive sketching application for the Microsoft Windows application with a transparent user interface.

The user interface designed for the *SketchPad for Windows* application is consistent with other Microsoft Windows applications. Since the user interface has now been defined, future work on *SketchPad for Windows* should be directed at improving the algorithm for entity recognition rather than on the user interface.

Efforts should also be made to implement the application on Pen Windows, the Microsoft Windows for pen based computers. Since the graphical user-interface for Pen Windows is consistent with that of Microsoft Windows 3.1, research in the area of implementing *SketchPad for Windows* on Pen Windows should be pursued. In order that *SketchPad for Windows* be easily extended to the pen based system, most of the sketching and editing features in the current version of *Sketchpad for Windows* have been implemented with the left mouse button. This greatly reduces the work for the future development team.

The future versions of *SketchPad for Windows* should support:

- * Recognition of spline entities and continuous strokes comprising lines and arcs.
- * Geometrical and spatial constraint intent capture.
- * A better interface for erasing entities.
- * Sketching on an isometric plane so as to support 3D object recognition. The sketched 3D objects should be recognized as standard part features. Commercially available CAD/CAM software development tool kits could be used to incorporate the features mentioned above.
- * Conversion of sketches into any CAD software either through IGES (Initial Graphics Exchange Specification) or DXF (Data Exchange Format).

Bibliography

- [Bally 87] J. M. Bally, "An Experimental View of the Design Process", System Design: Behavioral Perspective on Designers, Tools, and Organizations, Rouse, W. B. and Boff, K. R., editors, North-Holland, New York, 1987, pp. 65-82.
- [Budd] T. Budd, "Thinking Object-Oriented ", An Introduction to Object-Oriented Programming, Addison-Wesley Publishing Company, 1991, pp. 4.
- [DeJong 83] P. S. DeJong, J. S. Rising, and M. W. Almfeldt, "Freehand drawing", Engineering Graphics- Communication, Analysis, Creative Design, Kendall/Hunt Publishing Company, Dubuque, Iowa, 1983, pp. 13.
- [Faison 91] T. Faison, "Objects and Classes", Borland C++ 3 Object-Oriented Programming, SAMS, Carmel, Indiana, 1991, pp. 62-66.
- [Fang 88] R. C. Fang and D. G. Ullman, *Free-hand: A sketching recognition system for conceptual mechanical design*, Report, Design Process Research Group, Oregon State University, Corvallis, OR 97331, 1988
- [Goldstein and Alger 92] N. Goldstein and J. Alger, "The Folklore of Object-Oriented Software Development ", Developing Object Oriented Software for Macintosh, Addison-Wesley Publishing Company, Inc., 1992, pp. 52.
- [Gulur 92] S. Gulur, Personal observations, CAD/CAM Division, Godrej and Boyce Mfg. Co. Ltd., India.
- [Herbert 87] D. Herbert, "Study Drawings in Architectural Design: Applications for CAD Systems", *Proceedings of the 1987 Workshop of the Association for Computer-Aided Design in Architecture (ACADIA)*, 1987.
- [Herling *et al.*, 94] D. Herling, D. G. Ullman, and B. D'Ambrosio, Private communications, Mechanical Engineering Department, Oregon State University, Corvallis, OR 97331, 1994.
- [Hwang 91] T-S. Hwang, *The Design Capture System: Capturing Back-of-The-Envelope Sketches*, PhD thesis, Department of Mechanical Engineering, Oregon State University, Corvallis, OR 97331, 1991.
- [Iwata *et.al.* 87] K. Iwata, N. Sugimura, and W. Lee, "Knowledge-based recognition of hand-written drawings for product modeling in machine design", *Expert Systems in Computer-Aided Design*, Gero, J. S., editor, North-Holland, The Netherlands, 1987, pp. 375-401.

[Jenkins *et.al.*, 92] D. L. Jenkins and R. R. Martin, "Applying constraints to enforce user's intention's in free-hand 2D sketches", *Intelligent Systems Engineering*, Autumn 1992, pp. 31-49.

[Kasturi 90] R. Kasturi, S. T. Bow, W. El-Masri, J. Shah, J. R. Gattiker, and U. B. Mokate, "A system for interpretation of line drawings", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 12, No. 10, 1990, pp. 978-992.

[Knowlton 77] K. W. Knowlton, R. A. Beauchemin, and P. J. Quinn, *Technical Freehand Drawing and Sketching*, McGraw-Hill Book Company, 1977, pp. 5-7.

[Lafore 93], R. Lafore, "Event-Driven Programming ", *Windows Programming Made Easy*, The Waite Group, Corte Madera, CA, 1993, pp. 65.

[Luzzader 75] W. J. Luzzader, "Fundamentals and techniques of communication graphics", *Innovative design with an introduction to design graphics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975, pp. 91-93.

[McCord 92] J. W. McCord, "Windows Programming Basics ", *Developing Windows Applications with Borland C++ 3.0*, SAMS, Carmel, IN, 1992, pp. 13.

[Rittel 73] H. W. Rittel and M. M. Weber, "Dilemmas in a general theory of planning", *Police Science*, Vol. 4, 1973, pp. 155-169.

[Ryan 90] D. L. Ryan, "Symbolic Sketching", *Technical Sketching and Computer Illustration*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990, pp. 14-28.

[Suffel 89] C. Suffel and G. N. Blount, "Sketch form data input for engineering component definition, *Geometric Reasoning*, Woodwark, J., editor, Oxford University Press, 1989

[Sutherland 63] I. E. Sutherland, *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, 1963.

[Ullman 90] D.G. Ullman, S. L. Wood, D. Craig, "The Importance of Drawing in the Mechanical Design Process", *Computer and Graphics*, Vol 14, No. 2, 1990 pp. 263-274.

[Ullman 92] D. G. Ullman, "The Conceptual Design Phase: Concept Generation", *The Mechanical Design Process*, McGraw-Hill, Inc., 1992, pp. 158-159.

[Ullman 93] D.G. Ullman, "The OREO model of the Mechanical Design Process and Product", unpublished draft, Oregon State University, Corvallis, OR 97331, 1993

[Waldron *et.al.*, 88] M. B. Waldron and K. J. Waldron, "Conceptual CAD tools for mechanical designers", *Proceedings of Computers in Engineering conference*, Patton, E. M., editor, Computers and Graphics, volume 2, 1988, pp. 203-209

[Walnum 94] C. Walnum, "The Object Windows Classes ", Object-Oriented Programming with Borland C++ 4.0, Que Corporation, Indianapolis, IN, 1994, pp. 13.

[Wood 95] S. L. Wood, *An Architecture for a Function Driven Mechanical Design Solution Library*, PhD thesis, Department of Mechanical Engineering, Oregon State University, Corvallis, OR 97330, 1995.