AN OPEN GRAPHICS LIBRARY (OPENGL) BASED TOOLBOX FOR BIOMEDICAL IMAGE DISPLAY AND PROCESSING

by

Mehmet Olcay Kılıç

B.S. in Electronics and Telecommunications Engineering
Istanbul Technical University, 1995

Submitted to the Institute of Biomedical Engineering
in partial fulfillment of the requirements
for the degree of
Master of Science
in
Biomedical Engineering

Boğaziçi University
September, 2001

AN OPEN GRAPHICS LIBRARY (OPENGL) BASED TOOLBOX FOR BIOMEDICAL IMAGE DISPLAY AND PROCESSING

APPROVED BY:

Assoc.Prof. Ahmet Ademoğlu (Thesis Supervisor)	[illegible]
Assoc.Prof. Mehmed Özkan	[illegible]
Assoc.Prof. Ayşın Baytan Ertüzün	[illegible]

DATE OF APPROVAL: September 28, 2001

ACKNOWLEDGMENTS

At first, I would like to thank my thesis supervisor Assoc.Prof. Ahmet Ademoğlu for his patience, sincerity and endless support. I acknowledge his support and guidance with great respect.

I would like to thank Assoc.Prof. Mehmed Özkan and Assoc.Prof. Ayşın Baytan Ertüzün for participating in my thesis committee.

In order to honour them, I would also like to thank my parents, Meliha, Mustafa, Rana, Tuncay, Dinçay and Zeynep Kılıç. Thank you for endless support and endless patience during my whole lifetime.

ABSTRACT

AN OPEN GRAPHICS LIBRARY (OPENGL) BASED TOOLBOX FOR BIOMEDICAL IMAGE DISPLAY AND PROCESSING

By the development of Computerized Tomography (CT) in the late 70's and Magnetic Resonance Imaging (MRI) at the beginning of 80's, three dimensional images of the human body were generated in terms of slices of 2-D images. The availability of 3-D images gives researchers a better morphological, relational, and functional assessment of the anatomical structures. An Open Graphics Library (OpenGL) based image display and processing toolbox, called 3DVIEW, has been developed for the 3-D visualization of human tissues using the MRI/CT data. The 3-D rendering and visualization are performed via the OpenGL library routines while the basic image processing routines and Windows based Graphics User Interface (GUI) are developed using the Borland Object Windows 2.0 programming language and Borland C++ Version 4.5 compiler. The toolbox is capable of displaying the MRI/CT images in 2-D and 3-D as well as performing basic image processing techniques such as filtering, edge enhancement, and histogram watching. The Seeded Region Growing Algorithm (SRGA) is also included for the segmentation and 3-D visualization of different tissues. The toolbox is aimed to be an interface to which someone studying biomedical images can add other functions to perform his/her routine analysis and visualization.

Keywords: OpenGL, Borland C++, ObjectWindows Programming, Image Processing, Segmentation.

ÖZET

BİYOMEDİKAL GÖRÜNTÜLERİ İŞLEMEK VE GÖRÜNTÜLEMEK İÇİN OLUŞTURULMUŞ OPENGL TABANLI BİR ARAÇ KUTUSU

1970' li yılların sonunda Bilgisayar Tomografi ve 1980' li yılların başında Manyetik Rezonans Görüntüleme alanlarındaki gelişmeler sonucu, insan vücudunun üç boyutlu görüntüleri, iki boyutlu görüntü dilimleri halinde elde edilmeye başlanmıştır. Üç boyutlu görüntüler sayesinde araştırmacılar, vücut yapılarını, şekilsel, ilişkisel ve fonksiyonel yönlerden daha iyi değerlendirebilirler. 3DVIEW isimli, OpenGL tabanlı görüntü işleme ve görüntüleme araç kutusu, MRI ve CT verilerini kullanarak insan dokularının görüntülenmesi için geliştirilmiştir. Üç boyutlu parçalama ve görüntüleme, OpenGL kütüphane fonksiyonları kullanılarak; temel görüntü işleme fonksiyonları ve Windows tabanlı grafiksel kullanıcı arayüzü ise Borland Object Windows 2.0 programlama dili ve Borland C++ Versiyon 4.5 derleyici kullanılarak geliştirilmiştir. Araç kutusu, MRI/CT görüntülerini iki boyutlu ve üç boyutlu olarak görüntüleyebilmekte, süzgeçleme, kenar iyileştirme ve histogram eşleme gibi temel görüntü işleme tekniklerini yerine getirebilmektedir. Değişik dokuların bölütlendirilmesi ve üç boyutlu olarak görüntülenmesi için ayrıca nokta başlangıçlı alan büyütme (SRG) algoritması dahil edilmiştir. Araç kutusu, biyomedikal görüntülerle ilgilenen kişilerin, başka fonksiyonlar ekleyerek, bu fonksiyonların incelenmesini ve görselleştirilmesini sağlayacak bir arayüz olarak amaçlanmıştır.

Anahtar Kelimeler: OpenGL, Borland C++, Nesnesel Programlama, Görüntü İşleme, Bölütlendirme.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES x
LIST OF TABLES xii
LIST OF SYMBOLS xiii
LIST OF ABBREVIATIONS xiv
1. INTRODUCTION 1
1.1 Thesis Outline 2
2. MATHEMATICAL ELEMENTS IN 2-D and 3-D WORLD 4
2.1 The Rectangular Coordinate System 4
2.2 Spherical Coordinate System 4
2.3 Relationship between the rectangular and spherical coordinate systems 5
2.4 2-D Rotation on 2-D coordinate system 6
2.5 Mathemetical 3-D Operations 8
2.5.1 Generalized Transformation Matrix for 3-D Operations 8
2.5.2 3-D Rotation 8
2.5.3 3-D Translation 10
2.5.4 3-D Scaling 11
2.5.5 3-D Reflection 11
2.6 Viewport Planning for 3-D Graphics 12
2.6.1 Viewport Mapping 13
3. OPENGL 14
3.1 OpenGL Programming 14
3.1.1 OpenGL Command Syntax 15
3.1.2 OpenGL as a State Machine 16
3.1.3 OpenGL Related Libraries 17
3.1.4 Include Files 18
3.2 State Management and Drawing Geometrical Objects 19

3.2.1 Rendering Points, Lines and Polygons 21
3.2.2 Basic State Management 24
3.2.3 Vertex Arrays 25
3.2.4 Enabling Arrays 26
3.2.5 Specifying Data for the Arrays 26
3.3 OpenGL Viewing 28
3.3.1 The Viewing Transformation 29
3.3.2 The Modeling Transformation 29
3.3.3 The Projection Transformation 30
3.3.4 The Viewport Transformation 30
3.4 General Purpose Transformation Commands 31
3.5 Modeling Transformations 32
3.5.1 Translation Transformation 33
3.5.2 Rotation Transformation 33
3.5.3 Scaling Transformation 34
3.6 Viewing Transformations 35
3.6.1 gluLookAt() Function 36
3.7 Projection Transformations 36
3.7.1 Perspective Projection 37
3.7.2 Orthographic Projection 38
3.8 Viewport Transformation 40
3.8.1 Viewing Volume Clipping 40
3.8.2 Defining the Viewport 40
4. BASIC IMAGE PROCESSING TECHNIQUES 41
4.1 Introduction to Image Processing 41
4.2 Point Processes 41
4.2.1 Image Negation 42
4.2.2 Thresholding 42
4.2.3 Brightness Adjustment 42
4.2.4 Histogram Equalization 43
4.3 Area Processes 43
4.3.1 Neighbourhood Averaging 44

4.3.2 Low Pass Filtering 44
4.3.3 Laplacian Edge Enhancement 44
4.4 2-D Segmentation 46
4.4.1 Seeded Region Growing Algorithm (SRGA) 46
4.4.1.1 Algorithm Description 46
4.5 3-D Segmentation 47
5. WINDOWS PROGRAMMING and OBJECT ORIENTED SOFTWARE IMPLEMENTATION 48
5.1 Windows Programming Principles 48
5.1.1 Working with Class Hierarchies 48
5.1.1.1 Using a class 48
5.1.1.2 Inheritance 49
5.1.1.3 Member Functions 49
5.1.2 Object Typology 50
5.1.2.1 Window Classes 51
5.1.2.2 Dialog Boxes Classes 51
5.1.2.3 Control Classes 51
5.1.2.4 Graphics Classes 52
5.1.2.5 Printing Classes 52
5.1.2.6 Modules and Applications Classes 52
5.1.2.7 Doc and View Classes 52
5.1.2.8 Miscellaneous Classes 52
5.2 Object Oriented Software Implementation 53
5.2.1 New Application Class Declaration 53
5.2.2 OwlMain and Run Function 54
5.2.3 New Windows Class Declaration 55
5.2.4 Response Table Declaration 58
5.2.5 Adding a Menu Resource to the Main Window 59
5.2.6 Changing The Current Menu Resource 61
5.2.7 Adding New Functions for New Algorithms 62
5.2.7.1 Adding a New Pop-up Menu to the Menu Resource . . 62
5.2.7.2 Adding a New Menuitem to the Menu Resource 63

5.2.8 Adding Decorations to the Main Window 64
5.2.9 Creating the Control Bar 65
5.2.10 New Dialog box class declaration 68
5.2.10.1 Adding a New Dialog Box 71
6. IMPLEMENTATION 73
6.1 Toolbox Construction 73
6.2 Inserting New Algorithms 76
7. DISCUSSIONS, CONCLUSIONS and FUTURE WORK 95
APPENDIX A. DEVELOPED SOFTWARE LISTING 98
REFERENCES 100

LIST OF FIGURES

Figure 2.1 Mathematical system representations. 5

Figure 2.2 2-D rotation with the angle $\theta 1$. 6

Figure 2.3 3-D rotation about the z axis, y axis and x axis respectively. 9

Figure 3.1 Examples for the concave polygons. 22

Figure 3.2 Examples for the convex polygons. 22

Figure 3.3 Polygon rendering by passing the GL POLYGON parameter to the glBegin function. 23

Figure 3.4 Set of points rendering by passing the GL POINTS parameter to the glBegin function. 23

Figure 3.5 OpenGL workflow. 29

Figure 3.6 Case 1: Firstly a rotation and secondly a translation is applied to the object. 32

Figure 3.7 Case 2: Firstly a translation and secondly a rotation is applied to the object. 33

Figure 3.8 Translation Transformation. 33

Figure 3.9 Rotation Transformation. 34

Figure 3.10 Scaling Transformation. 35

Figure 3.11 Default camera position. 37

Figure 3.12 Perspective Viewing Volume. 38

Figure 3.13 Orthographic Viewing Volume. 39

Figure 5.1 BRW, requesting a new resource. 61

Figure 5.2 BRW, creating a new MENU resource.. 62

Figure 5.3 BRW, inserting a new Pop-up Menu Resource to the toolbox. 63

Figure 5.4 BRW, inserting a new Menu-item resource to the toolbox. 64

Figure 5.5 BRW, creating a new BITMAP resource for a button gadget. 66

Figure 5.6 BRW, constructing a new BITMAP resource for a button gadget. 67

Figure 5.7 BRW, creating a new dialog box resource. 71

Figure 5.8 BRW, constructing a new dialog box resource. 72

Figure 6.1 Constituted Window by using C++ Object Windows 2.0. 74

Figure 6.2	Preferences, Directory Path Definitions menu-item.	75
Figure 6.3	Preferences, Directory Path Definitions Dialog Box.	76
Figure 6.4	Slice pop up menu and its menuitems.	77
Figure 6.5	Open Slice Common Dialog Box.	78
Figure 6.6	Save Slice Common Dialog Box.	79
Figure 6.7	Processing pop up menu and its menuitems.	80
Figure 6.8	Original slice for the evaluation of processing results.	81
Figure 6.9	Histogram equalization process result.	82
Figure 6.10	Thresholding input dialog Box.	83
Figure 6.11	Thresholding process result.	84
Figure 6.12	2-D segmentation seed count parameter representation.	85
Figure 6.13	2-D segmentation seed representation.	86
Figure 6.14	2-D segmentation process result.	87
Figure 6.15	3-D segmentation process startup.	88
Figure 6.16	3-D segmentation seed representation.	89
Figure 6.17	3-D segmented region view, white matter.	90
Figure 6.18	Slice by Slice menu-item and result.	91
Figure 6.19	3-D zooming menu and result.	92
Figure 6.20	3-D Rotation menu and result.	93
Figure 6.21	3D view of 20-100 slice interval.	94

LIST OF TABLES

Table 3.1	OpenGL specific data type table. Corresponding ANSI C data types are also given in the table.	16
Table 3.2	OpenGL specific buffers. Their corresponding enumarated mask values are also given in the table.	20
Table 3.3	OpenGL geometric primitive names. OpenGL can easily render these geometric primitives.	24

LIST OF SYMBOLS

A_j	j th Seed Region
$L(f(x,y))$	Laplacian of the function $f(x,y)$
$N(x)$	Immediate neighbours of pixel x
T	Set of pixels that border at least one A_j
$\delta(x)$	Intensity Similarity value of pixel x

LIST OF ABBREVIATIONS

2-D	Two Dimensional
3-D	Three Dimensional
3DVIEW	The Name of the Developed Programme
BRW	Borland Resource Workshop
C	Current Modelview Matrix
CT	Computerized Tomography
GLUT	Graphics Library Utility Toolkit
M	Transformation Matrix
MRI	Magnetic Resonance Imaging
RGBA	Red, Green, Blue, Alpha
SRG	Seeded Region Growing
SRGA	Seeded Region Growing Algorithm
SSL	Sequentially Sorted List

1. INTRODUCTION

Today, imaging modalities such as CT and MRI are very common almost in every medical center. 3-D images are obtained by using these modalities. These images are mostly being used for pre-surgical, research, educational or post-surgery purposes.

The objective of this work is to create a toolbox for 2-D and 3-D display and visualization of tissues using MRI/CT data. A researcher may want to view these biomedical images in two or three dimensions slice by slice or in addition to basic image processing techniques, may want to add some new processing algorithms for the algorithm performance evaluation. 3DVIEW provides a base platform for these operations.

The toolbox consists of three main parts:

1) Windows programming part,
2) OpenGL programming part,
3) Processing part.

Menu driven graphical user interface resources have been created by the resource generator and compiler, BRW (Borland Resource Workshop) and whole code has been written in C/C++ programming language using Object Windows 2.0. For the compilation, Borland 32-bit C++ compiler has been used.

OpenGL is a powerful software interface to the graphics hardware and it is basically a library of functions for drawing, manipulating and rendering 2-D or 3-D shapes. This graphics library is responsible for the drawing part of the toolbox. The toolbox makes it possible to display and process 3-D biomedical images by taking advantage of OpenGL. The OpenGL is created for computer graphics applications and it facilitates the matrix calculations and matrix operations. This leads to a remarkable enhancement in the software performance. The user may view the related slice in 2-D or 3-D manner or may display some set of slices in 3-D. The toolbox can render some portion or the whole biomedical volume and can make it possible to apply some 3-D operations such as 3-D rotations in x, y and z coordinates and 3-D zooming.

For the processing part, basic image processing algorithms such as histogram equalization, image negation, brightness adjustment, thresholding, filtering, edge enhancement, 2-D and 3-D segmentation utility have been implemented. All these functions are in modular structure and new menu-item or algorithm functions can be inserted into the toolbox by making some modifications on the software.

1.1 Thesis Outline

Chapter 1 provides an introduction to the problem.

Chapter 2 gives detailed information about the mathematical world of 2-D and 3-D graphics.

Chapter 3 deals with the all aspects of OpenGL including the OpenGL principles, specific OpenGL commands or functions and the OpenGL interior world.

Chapter 4 presents detailed information about the image processing techniques and algorithms realized in this study.

Chapter 5 provides a basic background of WINDOWS programming concept with some object oriented coding samples realized in the toolbox. This chapter also explains the steps should be taken for inserting a new Windows resource or adding a new algorithm to the toolbox.

Chapter 6 presents the results of toolbox implementation in detail.

The discussions, conclusions and the recommendations for future work are given in Chapter 7.

2. MATHEMATICAL ELEMENTS IN 2-D and 3-D WORLD

There are two common ways to represent points in the real world: The Rectangular Coordinate System and the Spherical Coordinate System [1].

2.1 The Rectangular Coordinate System

In this representation, points are identified by ordered triples (x, y, z) of numbers with X, Y and Z coordinates providing the directed distance from the point to the YZ, XZ and XY planes respectively.

2.2 Spherical Coordinate System

In this system, each point is represented by an ordered triple of numbers (D, θ, ϕ). The distance from the point P to the origin is D. The angle between the line OP and the positive direction of the Z axis is ϕ, the angle between the positive direction of the X axis and the line OP' (projection of OP onto the XY plane) is θ. This angle is measured in counterclockwise direction from the X axis.

These system representations are given in Figure 2.1

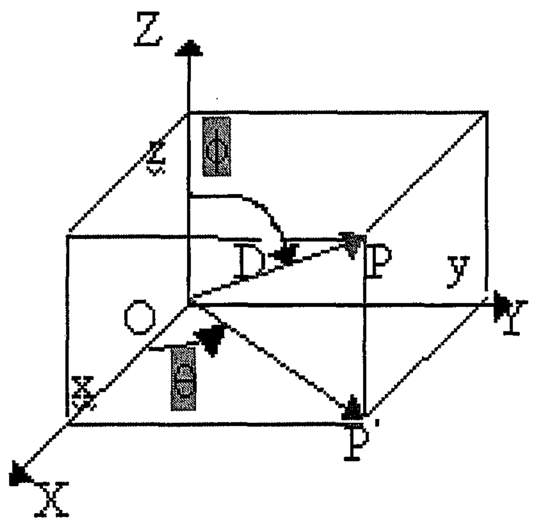


Figure 2.1 Mathematical system representations.

2.3 Relationship between the rectangular and spherical coordinate systems

Since both rectangular and spherical coordinate systems are being used in computer graphics, it is often necessary to convert between the two systems. From trigonometry, we can establish the following relationships between two coordinate systems.

$$x = D \cdot \sin\phi \cdot \cos\theta \tag{2.1}$$

$$y = D \cdot \sin\phi \cdot \sin\theta \tag{2.2}$$

$$z = D \cdot \cos\phi \tag{2.3}$$

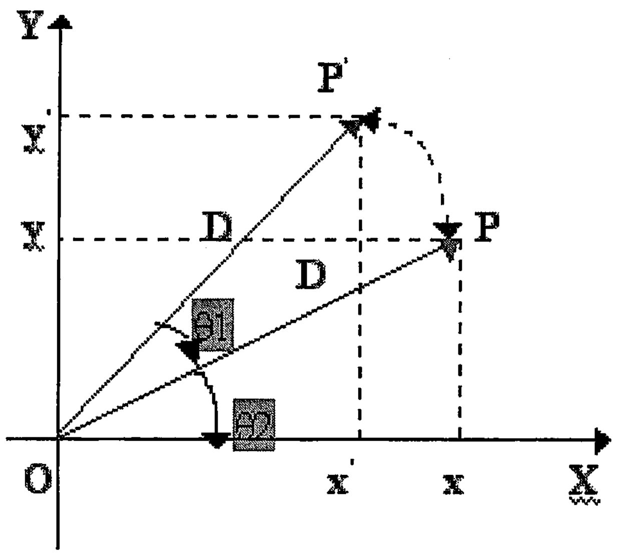


Figure 2.2 2-D rotation with the angle $\theta 1$.

and

$$D = \sqrt{(x^2 + y^2 + z^2)} \quad (2.4)$$

$$\theta = \arctan(y/x) \quad (2.5)$$

$$\phi = \arccos(z/D) \quad (2.6)$$

2.4 2-D Rotation on 2-D coordinate system

The effect of rotation of point P on two dimensional coordinate system is shown in Figure 2.2. $\theta 1$ is the rotation angle.

We obtain the following equations:

$$x = D \cdot \cos\theta 2 \tag{2.7}$$

$$x' = D \cdot \cos(\theta 1 + \theta 2) \tag{2.8}$$

$$x' = D \cdot \cos(\theta 1) \cdot \cos(\theta 2) - D \cdot \sin(\theta 1) \cdot \sin(\theta 2) = x \cdot \cos(\theta 1) - y \cdot \sin(\theta 1) \tag{2.9}$$

and,

$$y = D \cdot \sin(\theta 2) \tag{2.10}$$

$$y' = D \cdot \sin(\theta 1 + \theta 2) \tag{2.11}$$

$$y' = D \cdot \sin(\theta 1) \cdot \cos(\theta 2) + D \cdot \cos(\theta 1) \cdot \sin(\theta 2) = y' = x \cdot \sin(\theta 1) + y \cdot \cos(\theta 1) \tag{2.12}$$

Therefore, for 2-D rotation operation, we obtain the following transformation matrix used in equation 2.13.

$$[x', y'] = [x, y] \begin{bmatrix} \cos\theta 1 & \sin\theta 1 \\ -\sin\theta 1 & \cos\theta 1 \end{bmatrix} \tag{2.13}$$

2.5 Mathemetical 3-D Operations

2.5.1 Generalized Transformation Matrix for 3-D Operations

The generalized 4 x 4 transformation matrix for a point in a 3-D homogeneous coordinate system is:

$$\begin{bmatrix} A & B & C & 0 \\ D & E & F & 0 \\ G & H & I & 0 \\ L & M & N & 1 \end{bmatrix}$$

and it can be partitioned into three sub-matrix:

$$\begin{bmatrix} I_{3\times3} & III_{3\times1} \\ & \\ II_{1\times3} & 1 \end{bmatrix}$$

Sub-matrix I allows us to perform scaling, reflection and rotation of 3-D objects. Sub-matrix II produces linear translation and Sub-matrix III permits us to combine the translation and the transformation in multiplication form [2].

2.5.2 3-D Rotation

Let us consider first a rotation about the z axis as it is shown in Figure 2.3. In this case, the z dimension does not change so the transformation matrix will have zeros in the third row and third column, except for unity on the main diagonal. The other terms are determined by the Equation 2.13 considering the rotation of the point in two dimensional coordinate system. As a matter of convention, we measure the rotation

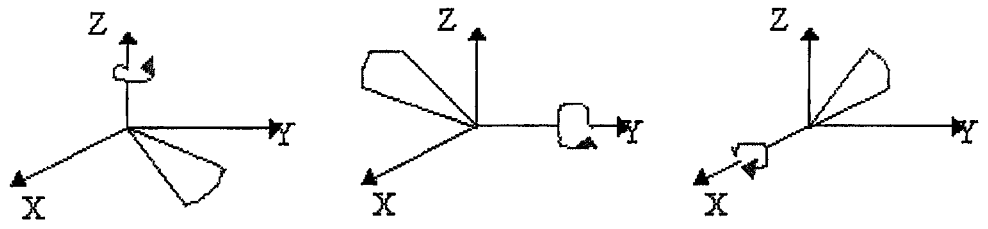


Figure 2.3 3-D rotation about the z axis, y axis and x axis respectively.

angle θ counterclockwise.

These leads to the following transformation matrix used in Equation 2.14 for a rotation of angle θ about the z axis:

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

For the rotation of angle θ about the Y axis,

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

equation is obtained.

The transformation matrix that realizes rotation of angle θ about the X axis takes it place in the equation shown below.

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & \text{-}\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

2.5.3 3-D Translation

The transformation that translates a point (x, y, z) to a new point (x', y', z') through (L, M, N) is expressed by the equation shown below:

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ L & M & N & 1 \end{bmatrix} \tag{2.17}$$

where L, M and N are the distances to be moved from x, y and z, respectively.

2.5.4 3-D Scaling

The scaling matrix:

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & E & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

This indicates that we can scale each of the coordinates individually. If the object is to be enlarged by a factor of two, we simply choose:

$$A = E = I = 2 \tag{2.19}$$

2.5.5 3-D Reflection

An object is reflected through a plane by manipulating the diagonal elements in the 3-D transformation matrix. This is provided by generating a mirror image of all points of the object on the opposite side of the plane. For an example, in a reflection through the XY plane, only the Z coordinate values will be affected, and these will be reversed in sign. Here is the transformation matrix for the reflection process mentioned above:

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.20}$$

Similarly, for the reflection through the YZ plane the equation is,

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.21}$$

and for a reflection through the XZ plane, it is:

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.22}$$

2.6 Viewport Planning for 3-D Graphics

Three dimensional viewport planning involves clipping the contents of the data against the boundaries of the desired view. This clipping process is to ensure that only the visible portions of the picture are transformed into plotting instructions for the display device [3].

If we want to specify a limited view volume, we simply introduce a front clipping plane and a back clipping plane. Both clipping planes are parallel to the view plane and both are specified by a distance along the view plane normal from the view reference point. In other words, these planes are assumed to be perpendicular to the line of sight.

2.6.1 Viewport Mapping

After clipping a picture against the view volume, the final step is to map it to the viewport. Then the actual screen coordinates on the physical device are determined [4].

3. OPENGL

The OpenGL graphics system is a software interface to graphics hardware [5]. The GL stands for Graphics Library. OpenGL is really a hardware independent specification of a programming interface and it is constituted by Silicon Graphics company.

3.1 OpenGL Programming

Before using the OpenGL, one should know how to program in the C programming language and should have some background in mathematics especially in geometry, trigonometry and linear algebra.

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive two or three dimensional applications.

OpenGL is designed as a hardware independent interface to be implemented on many different hardware platforms. OpenGL doesn't provide high level commands for describing models of three dimensional objects. With OpenGL, you must build up your desired model from a set of geometric primitives points, lines, and polygons.

Basic steps for writing applications on OpenGL are:

- Constructing shapes from geometric primitives, thereby creating mathematical descriptions of objects. OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.
- Arranging the objects in three dimensional space and selecting the desired point for viewing the composed scene.
- Calculating the color of all the objects.

- Converting the mathematical description of objects and their associated color information to pixels on the screen. This process is called rasterization.

During these stages, OpenGL might perform other operations, such as eliminating parts of objects that are hidden by other objects. Rendering is the process by which a computer creates images from models. These models are constructed from geometric primitives namely points, lines, and polygons that are specified by their vertices. The final rendered image consists of pixels drawn on the screen. A pixel is the smallest visible element the display hardware can put on the screen.

Information about the pixels for instance, what color they are supposed to be is organized in memory into bitplanes. A bitplane is an area of memory that holds one bit of information for every pixel on the screen. The bitplanes are themselves organized into a framebuffer, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen.

3.1.1 OpenGL Command Syntax

OpenGL commands use the prefix gl and initial capital letters for each word making up the command name (glClearColor(), for example). Similarly, OpenGL defined constants begin with GL, and use all capital letters and use underscores for separating the words like GL_ .

Sometimes, seemingly extraneous letters are appended to some command names for example, the 3f in glColor3f function. In particular, the 3 part of the suffix indicates that three arguments are given and the f part of the suffix indicates that the arguments are floating point numbers.

Some OpenGL commands accept as many as eight different data types for their arguments and these data types are defined in Table 3.1.

Table 3.1

OpenGL specific data type table. Corresponding ANSI C data types are also given in the table.

Suffix	Data Type	Corresponding ANSI C Data Type	OpenGL Type
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint
s	32-bit floating	float	GLfloat
d	64-bit floating	double	GLdouble
ub	8-bit unsigned integer	unsigned char	GLubyte
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned long	GLusint

Thus, the two commands, glVertex2i(1,3) and glVertex2f(1.0,3.0) are equivalent, except that the first one specifies the vertex coordinates as 32 bit integers, and the second specifies them as floating numbers.

Some OpenGL commands can take a final letter v, which indicates that the command takes a pointer to a vector of values rather than a series of individual arguments as it is in the example shown below:

```
Glfloat color_array[]={1.0, 0.0, 0.0}

GlColor3fv(color_array);
```

3.1.2 OpenGL as a State Machine

OpenGL is a state machine. You put it into various states (or modes) that then remain in effect until you change them. For example, you can set the current color to white, red, or any color and every object is drawn with that color until you set the current color something else. The current color is only one of many state variables that OpenGL maintains. Polygon drawing modes, positions and characteristics of lights, and material properties of the objects, viewing and projection transformations are the examples for the states that OpenGL maintains. Many state variables refer to modes that are enabled or disabled with the command glEnable or glDisable.

Each state variable or mode has a default mode, and any point you can query the system for each variable's current value. Typically, you use one of the six folowing commands to do this:

- glGetBooleanv(),
- glGetDoublev(),
- glGetFloatv(),
- glGetIntegerv(),
- glGetPointerv(),
- glIsEnabled().

3.1.3 OpenGL Related Libraries

OpenGL provides a powerful but primitive set of rendering commands, and all higher level drawing must be done in terms of these commands. A number of libraries exist to allow you to simplify your programming tasks, including the following:

The OpenGL Utility Library (GLU) contains several routines that use lower level OpenGL commands to perform such tasks as setting up the specific matrix for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. GLU routines use the prefix glu.

For every Window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix glX. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix wgl. For IBM OS/2, the PGL is the Presentation Manager to OpenGL interface, and its routines use the prefix pgl.

The OpenGL Utility Toolkit (GLUT) is a window system independent toolkit to hide the complexities of differing window systems APIs. GLUT routines use the prefix glut.

3.1.4 Include Files

Every OpenGL source file begins with include <.../gl.h> command.

If we are directly accessing a window interface library to support OpenGL, such as GLX, AGL, PGL or WGL, you must include additional header files. For example, if you are calling GLX, you may need to add that line to your code:

include <.../glx.h>

and if you are using GLUT for managing your window manager tasks, you should include

include <.../glut.h>

Note that since the glut.h includes gl.h, glu.h and glx.h automatically, including three files is redundant.

As it is mentioned before, OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboards or mouse.

GLUT makes it possible to deal with the Window Management. However, since the 3DVIEW is based on Object Oriented Windows Programming principles, it is not needed to use GLUT library.

3.2 State Management and Drawing Geometrical Objects

In OpenGL, unless you specify otherwise, everytime you issue a drawing command, the specified object is drawn. In some other systems, you first make a list of things to draw and when your list is complete, you tell the graphics hardware to draw the items. The first style is called immediate mode graphics and is the default OpenGL style.

Before drawing something, it is necessary to clear the window entirely. The color you use for the background depends on the application. For a word processor, you might clear to white. As it is mentioned before, the colors of pixels are stored in the graphics hardware known as bitplanes. There are two methods of storage. Either the red, green, blue and alpha (RGBA) values of a pixel can be directly stored in the bitplanes, or a single index value that references a color look up table is stored. As an example, these lines of code clear an RGBA mode window to black.

```
glClearColor(0.0, 0.0, 0.0, 0.0);

glClear(GL_CLEAR_BUFFER_BIT);
```

The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color. The single parameter to glClear() indicates which buffer is to be cleared. These masks are represented in Table 3.2. In this case the code only clears the color buffer. For example, to clear both the color buffer and the depth buffer, you would use the following sequence of commands.

```
glClearColor(0.0, 0.0, 0.0, 0.0);

glClearDepth(1.0);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Typically, you set the clearing color once, early in your application, and then you clear the buffers as often as necessary. OpenGL keeps track of the current clearing color as a state variable rather than requiring you to specify it each time a buffer is cleared.

Command form : void glClearColor(red, green, blue, alpha);

Command form : void glClear(mask);

Table 3.2

OpenGL specific buffers. Their corresponding enumarated mask values are also given in the table.

Buffer	Mask Name
Color Buffer	GL_COLOR_BUFFER_BIT
Depth Buffer	GL_DEPTH_BUFFER_BIT
Accumulation Buffer	GL_ACCUM_BUFFER_BIT
Stencil Buffer	GL_STENCIL_BUFFER_BIT

Before issuing a command to clear multiple buffers, you have to set the values to which each buffer is to be cleared by commands shown below:

- glClearColor(),
- glClearDepth(),
- glClearAccum(),
- glClearIndex(),

- glClearStencil().

3.2.1 Rendering Points, Lines and Polygons

This section explains how to describe OpenGL geometric primitives. All geometric primitives are eventually described in terms of their vertices and coordinates that define the points themselves.

One difference comes from the limitations of computer based calculations. In any OpenGL implementation, floating point calculations are of finite precision, and they have round off errors. Another important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/100 of an inch wide, they are still far from the mathematician's concepts related with infinity. In OpenGL, many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

A point is represented by a set of floating point numbers called the vertex. Vertices specified by the user as two dimensional (that is, with only x and y coordinates) are assigned as z coordinate equal to zero by OpenGL. In OpenGL, the term line refers to a line segment. Polygons are the areas enclosed by single closed loops of line segments. OpenGL makes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons can not intersect. Second, OpenGL polygons must be convex. Concave and convex polygon samples are illustrated in Figure 3.1 and Figure 3.2 respectively.

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. You use the glVertex() command to specify a vertex.

Void glVertex{234}{sifd}{v}{coords};

Calls to glVertex() function are only effective between a glBegin() and glEnd()

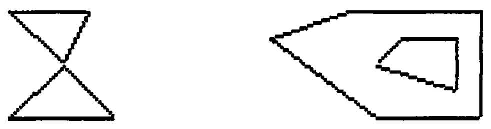


Figure 3.1 Examples for the concave polygons.

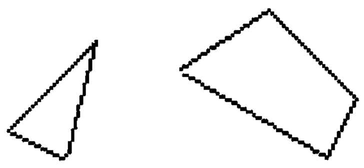


Figure 3.2 Examples for the convex polygons.

pair. You can supply up to four coordinates (x, y, z, w) for a particular vertex or as few as two (x, y) by selecting the appropriate version of the command. If you use a version that does not specify z or w, z is understood to be 0 and w is understood to be 1. Some examples are given below.

glVertex2s(2, 3); (Z coordinate is assumed as 0)

glVertex3d(0.0, 0.0, 3.1415); (Double precision floating point representation)

glVertex4f(2.3, 1.0, 2.2, 2.0); (Euclidean form, all coordinates are divided by 2)

In order to create a set of points, a line, or a polygon from those vertices you bracket each set of vertices between a call to glBegin() and a call to glEnd(). The argument passed to glBegin() determines what kind of geometric primitive is constructed from these vertices.

glBegin(GL_POLYGON);

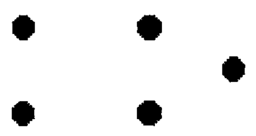


Figure 3.3 Polygon rendering by passing the GL POLYGON parameter to the glBegin function.

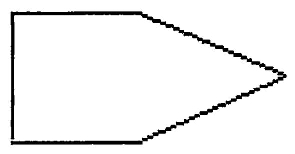


Figure 3.4 Set of points rendering by passing the GL POINTS parameter to the glBegin function.

```
        glVertex2f(0.0, 0.0);

        glVertex2f(0.0, 3.0);

        glVertex2f(4.0, 3.0);

        glVertex2f(6.0, 1.5);

       glVertex2f(4.0, 0.0);

glEnd();
```

The graphical results are represented in Figure 3.4 and Figure 3.3 respectively.

The function void glBegin(mode) activates the primitive rendering. The type of primitive is indicated by the mode parameter, which can be any of the values shown in Table 3.3. The function void glEnd(); function marks the end of a vertex data list.

Instead of calling glVertex() functions, vertex arrays can be used in order to enhance the software performance.

Table 3.3

OpenGL geometric primitive names. OpenGL can easily render these geometric primitives.

Value	Meaning
GL_ POINTS	Individual Points
GL_ LINES	Pair of vertices interpreted as line segments
GL_ LINE_ STRIP	Series of connected line segments
GL_ LINE_ LOOP	Same as above with segment between last and first vertex
GL_ TRIANGLES	Triples of vertices are interpreted as a triangle
GL_ TRIANGLE_ STRIP	Linked strip of triangles
GL_ QUADS	Quadruples of vertices are interpreted as a four sided polygon
GL_ QUAD_ STRIP	Linked strip of Quadrilaterals
GL_ POLYGON	Convex polygon

3.2.2 Basic State Management

OpenGL maintains many states and state variables. By default, most of these states are initially inactive. To turn on and turn off many of these states, two simple commands are being used:

```
void glEnable(capability);

void glDisable(capability);
```

glEnable() function turns on a capability, and glDisable() turns it off. There are over 40 enumerated values that can be passed as a parameter to glEnable() or glDisable(). The function glIsEnabled(capability) returns GL_ TRUE or GL_ FALSE, depending upon whether the capability is currently activated or not. The states have two settings: on and off.

By default, a point is drawn as a single pixel on the screen, a line is drawn solid and one pixel wide, and polygons are drawn solidly filled in.

In order to control the size of a rendered point, glPointSize() function is used and the desired size in pixels is passed as the argument.

void glPointSize(Glfloat size); (size range is (0.0 ; 1.0) and by default it is 1.0)

3.2.3 Vertex Arrays

Drawing a 20 sided polygon requires 22 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd(); These function calls have a great deal of overhead and can hinder performance. An additional problem is the redundant processing of vertices that are shared between adjacent polygons.

OpenGL has vertex array routines that allow you to specify a lot of vertex related data with just a few arrays and to access that data with equally few function calls. Using vertex array routines, all 20 vertices in a 20 sided polygon could be put into one array and called with one function. Arranging data in vertex arrays increases the performance of the application. In 3DVIEW, vertex arrays were generated and vertex array functions were used in order the enhance the overall performance. This technique also makes the software more friendly by reducing the code size. Also, it allows non redundant processing of shared vertices.

There are three steps to using vertex arrays to render geometry:

1) Enabling up to six arrays, each to store a different type of data. The arrays are vertex coordinates, RGBA colors, color indices, surface normal, texture coordi-

nates, or polygon edge flags.

2) Putting data into the array or arrays. The arrays are accessed by the addresses of (that is, pointers to) their memory locations.

3) Drawing geometry with the data. OpenGL obtains the data from all activated arrays by using the pointers.

Interleaved vertex array data is another common method of organization. Instead of having up to six different arrays, each maintaining a different type of data, you might have different type of data mixed into a single array.

3.2.4 Enabling Arrays

void glEnableClientState (array name) activates the chosen array. In 3DVIEW toolbox, vertex array and color array were activated by this function call shown below.

glEnableClientState();

void glDisableClientState(array name) function specifies the array to disable.

3.2.5 Specifying Data for the Arrays

void glVertexPointer(size, type, stride, *pointer);

The function shown above specifies where spatial coordinate data can be accessed. Pointer is the memory address of the first coordinate of the first vertex in the array. Type specifies the data type (GL_ SHORT, GL_ INT, GL_ FLOAT, or GL_ DOUBLE) of each coordinate in the array. Size is the number of coordinates per vertex, which must be 2, 3, or 4. Stride is the byte offset between consecutive vertices. If stride is 0, the vertices are understood to be tightly packed in the array.

Except glVertexPointer() function, there are five similar functions:

1) void glColorPointer(size, type, stride, *pointer);

2) void glIndexPointer(type, stride, *pointer);

3) void glNormalPointer(type, stride, *pointer);

4) void glTexCoordPointer(size, type, stride, *pointer);

5) void glEdgeFlagPointer(stride, *pointer);

Using a stride of other than zero can be useful, especially when dealing with interleaved arrays.

void glDrawArrays(mode, first, count);

glDrawArrays function constructs a sequence of geometric primitives using array elements starting at first and ending at first+count−1 of each enabled array. Mode specifies what kinds of primitives are constructed.

It is a fact that there is always a trade off between the display speed and the quality of the image. If there are millions of points, it probably looks good but might take a long time to render. If the image consists of small number of geometric primitives, it renders quickly but might have a jagged appearence.

3.3 OpenGL Viewing

This section explains how to use OpenGL to accomplish these tasks: how to position and orient models in three dimensional space and how to establish the location of the viewpoint. A series of three computer operations convert an object's three dimensional coordinates to pixel positions on the screen.

1) Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, you use a combination of several transformations to draw a scene.
2) Since the scene is rendered on a rectangular window, objects that lie outside the window must be clipped. In three dimensional computer graphics, clipping occurs by throwing out object on one side of a clipping plane.
3) Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as viewport transformation.

This flow is summarized in Figure 3.5. In the software code, the viewing transformations must precede the modeling transformations but the projection and viewport transformations can be specified at any point before drawing occurs.

Modeling and viewing transformations are combined to form the modelview matrix, which is applied to the incoming object coordinates to yield eye coordinates. Next, OpenGL applies the projection matrix to yield clip coordinates. This transformation defines a viewing volume; objects outside this volume are clipped so that they are not drawn in the final scene. Finally, the transformed coordinates are converted to window coordinates by applying the viewport transformation. You might correctly suppose that the x and y coordinates are sufficient to determine which pixels need to be drawn on the screen. However, all the transformations are performed on the z coordinate as well. Suppose that two vertices have the same x and y values but different

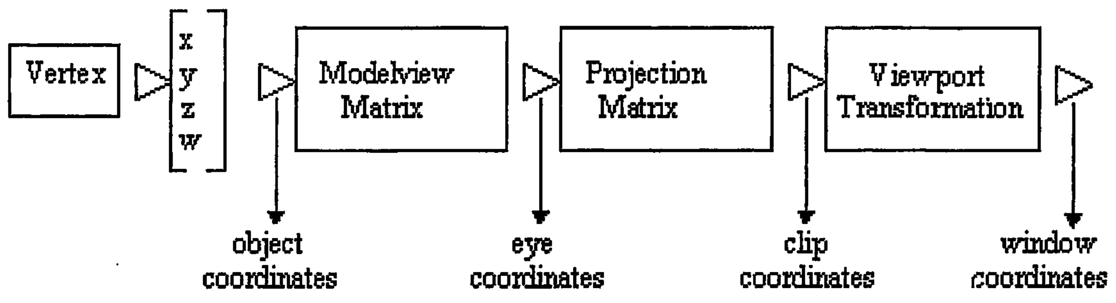


Figure 3.5 OpenGL workflow.

z values. OpenGL can use this information to determine which vertex is obscured by the other and can avoid drawing the hidden vertex (hidden surface removal or z buffer algorithm).

3.3.1 The Viewing Transformation

Before the viewing transformation can be specified, the current matrix is set to the identity matrix with glLoadIdentity() function call. This step is necessary since most of the transformation commands multiply the current matrix by the specified matrix and then set the result to be the current matrix. The viewing transformation is specified with gluLookAt() function. The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up. If gluLookAt() was not called, the camera has a default position and orientation. By default the camera is situated at the origin, points down the negative z axis. The command gluLookAt() encapsulates a series of rotation and translation commands.

3.3.2 The Modeling Transformation

You use the modeling transformation to position and orient the model. For example, you can rotate, translate, or scale the model or perform some combination of these operations. For example glScalef() is the modeling transformation for enlarging the object in any dimension. If all the arguments are 1.0, this command has no effect. Instead of moving the camera with a viewing transformation it is possible to the cube away from the camera with a modeling transformation. This is also why modeling and viewing transformations are combined to modelview matrix.

3.3.3 The Projection Transformation

Specifying the projection transformation is like choosing a lens for a camera. You can think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. This is equivalent to choosing among wide angle, normal, and telephoto lenses, for example.

In addition, the projection transformation determines how object are projected onto the screen. Two basic types of projections are provided in OpenGL. One type is perspective projection, which matches how you see thing in daily life. Perspective makes objects that are far from you appear smaller. The command glFrustum() is used for perspective vision in OpenGL. The other type of projection is orthographic, which maps object directly onto the screen without affecting their relative size. Therefore, the final image reflects the measurements of objects rather than how they might look. In order to set the projection transformation, the OpenGL command glMatrixMode() is performed with the argument GL_ PROJECTION. This indicates that the current matrix specifies the projection transformation.

3.3.4 The Viewport Transformation

Together, the viewport and the projection transformation determine how a scene gets mapped onto the computer screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped. Since the viewport specifies the region the image occupies on the screen, you can think of the viewport transformation as defining the size and the location of the final processed photograph.

The arguments to glViewport describe the origin of the available screen space within the window. The width and the height of the screen area, all measured in pixels are determined with this transformation. If the window's size changes, the viewport needs to change accordingly.

3.4 General Purpose Transformation Commands

As described in the preceding section, it is necessary to state whether you want to modify the modelview or projection matrix before supplying a transformation command. The command glMatrixMode() is used for that purpose.

Void glMatrixMode(mode); specifies whether the modelview, projection matrix will be modified, using the argument GL_ MODELVIEW, GL_ PROJECTION for mode. Subsequent transformation commands affect the specified matrix. Note that only one matrix can be modified at a time. By default, the modelview matrix is the one that is modifiable and every matrix contain the identity matrix. With the command glLoadIdentity(), it is possible to clear the currently modifiable matrix for future transformation commands. It is necessary to call this command before specifying protection or viewing transformations, but it can also be used before specifying a modeling transformation.

Void glLoadIdentity(void); command sets the currently modifiable matrix to the 4x4 identity matrix. Void glMultMatrix(*m); command multipies the matrix specified by the sixteen values pointed to by m by the current matrix and stores the result as the current matrix.

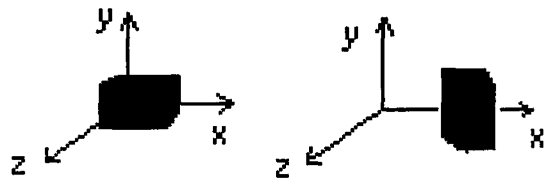


Figure 3.6 Case 1: Firstly a rotation and secondly a translation is applied to the object.

All viewing and modeling transformations are represented as 4x4 matrix. Each successive transformation command or glMultMatrix() command multiplies a new 4x4 matrix M by the current modelview matrix C to yield CM. Finally, vertices v are multiplied by the current modelview matrix (CMv).

3.5 Modeling Transformations

Viewing and modeling transformations are combined into a single modelview matrix. Let's think about transformations. A 90 degree counterclockwise rotation about the origin around the z axis, and a translation down the x axis. Figure 3.6 shows this case. If we translate it down the x axis first and then rotate about the origin the object takes another place in real world coordinates. Figure 3.7 shows the second case. If you do transformation A and then transformations B, you almost always get something different than if you do them in the opposite order.

Three OpenGL routines for modeling transformations are glTranslate(), glRotate, and glScale(). All these three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling glMultMatrix() with that matrix as the argument. However, these three routines might be faster than using glMultMatrix() command.

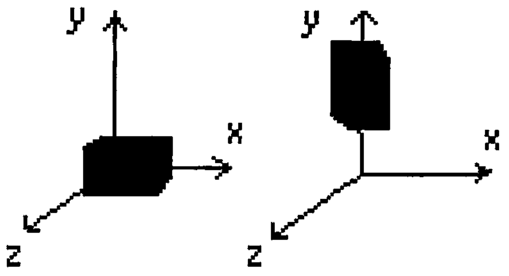


Figure 3.7 Case 2: Firstly a translation and secondly a rotation is applied to the object.

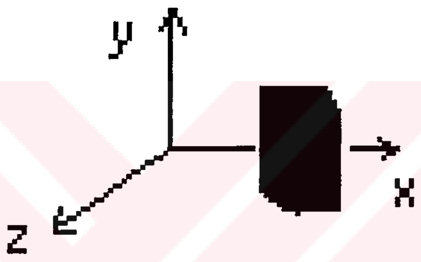


Figure 3.8 Translation Transformation.

3.5.1 Translation Transformation

Void glTranslate(x, y, z); moves the local coordinate system by the values specified as arguments. Figure 3.8 shows the effect of glTranslate(3, 0, 0); command.

3.5.2 Rotation Transformation

Void glRotate(angle, x, y, z); function multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies

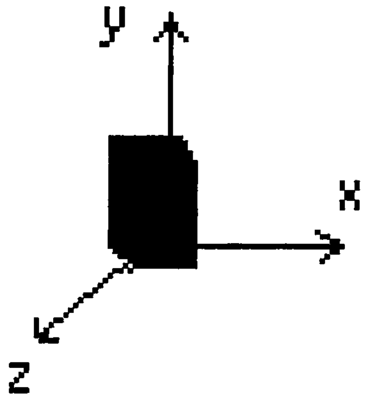


Figure 3.9 Rotation Transformation.

the angle of rotation in degrees. The effect of glRotate(90.0, 0.0, 0.0, 1.0), which is a rotation of 90 degrees about the z axis, is shown in Figure 3.9.

3.5.3 Scaling Transformation

Void glScale(x, y, z); function multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axis. Each x, y, z coordinate of every point in the object is multiplied by the corresponding argument x, y, z. With the local coordinate system aproach, the local coordinate axis are stretched, shrunk or reflected by the x, y, and z factors. Figure 3.10. shows the effect of glScale(0.5, -0.5, 0.0) transformation operation.

The command glScale() is the only modeling transformation command that changes the size of an object.

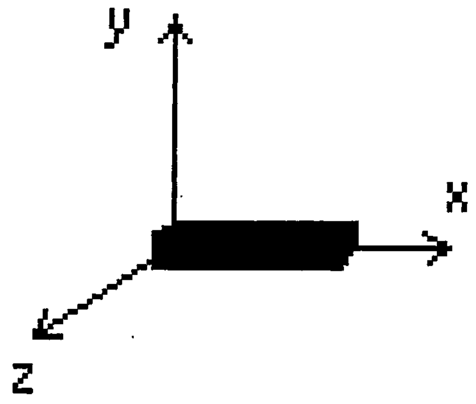


Figure 3.10 Scaling Transformation.

3.6 Viewing Transformations

A viewing transformation changes the position and the orientation of the viewpoint. It positions the camera tripod, pointing the camera toward to the model. Viewing transformations are generally composed of translations and rotations. A modeling transformation that rotates an object counterclockwise is equivalent to a viewing transformation that rotates the camera clockwise. The viewing transformation commands must be called before any modeling transformations are performed.

Since the viewpoint is initially located at the origin and since objects are often constructed there as well, in general you have to perform some transformation so that the object can be viewed.

In the simplest case, you can move the viewpoint backward, away from the objects; this has the same effect as moving the objects away from the viewpoint.

3.6.1 gluLookAt() Function

Programmers construct a scene around the origin or some other convenient location, then they want to look at it from an arbitrary point to get a good view of it. The command gluLookAt() is designed for just this purpose. It takes three sets of arguments, which specify the location of the viewpoint, define a reference point toward which the camera is aimed, and indicate which direction is up. Choose the viewpoint to yield the desired view of the scene. The reference point is typically somewhere in the middle of the scene. It might be a little trickier to specify the correct up vector.

Void gluLookAt(eyex, eyey, eyez, centerx, centerx, centerx, upx, upy, upz)

The command shown above defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified bye eyex, eyey, and eyez. The centerx, centery, and center z arguments specify any point along the desired line of sight, but typically they are some point in the center of the scene being look at. The upx, upy, and upz arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

In the default position, the camera is at the origin, looking down the negative z axis, and has the positive y axis as straight up as it is shown in Figure 3.11. This is the same as calling

gluLookAt (0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, 1.0, 0.0); function.

3.7 Projection Transformations

This section explains how to define the desired projection matrix, which is also used to transform the vertices in your scene.

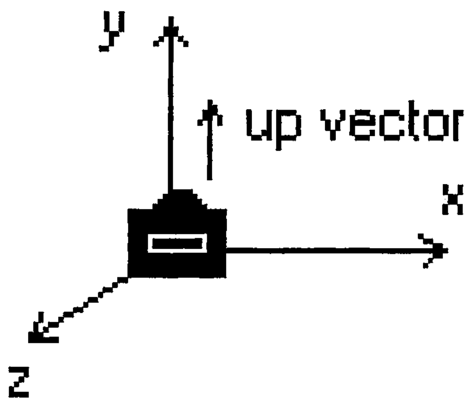


Figure 3.11 Default camera position.

glMatrixMode(GL_ PROJECTION);

glLoadIdentity();

commands are issued so that the commands affect the projection matrix rather than the modelview matrix. The purpose of the projection transformation is to define a viewing volume, which is used in two ways. The viewing volume determines how an object is projected onto the screen(that is, by using a perspective or an orthographic projection), and it defines which portions of the objects are clipped out of the final image.

3.7.1 Perspective Projection

The farther an object is from the camera, the smaller it appears in the final image. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid. Objects that fall within the viewing volume are projected toward the apex of the pyramid, where the camera or viewpoint is. Objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away, in the larger part of the frustum.

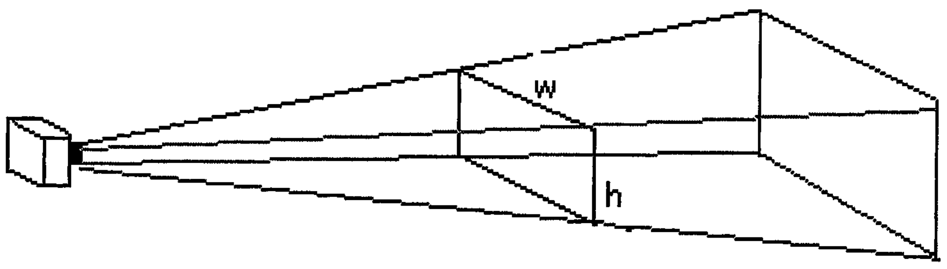


Figure 3.12 Perspective Viewing Volume.

Perspective view is illustrated in Figure 3.12.

Void glFrustum(left, right, bottom, top, near, far); command creates a matrix for a perspective view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (left, bottom, near) and (right, top, near) specify the (x, y, z) coordinates of the lower left and upper right corners of the near clipping plane; near and far give the distances from the viewpoint to the near and far clipping planes.

3.7.2 Orthographic Projection

With an orthographic projection, the viewing volume is a rectangular, or more informally, a box shown in Figure 3.13.

The size of the viewing volume does not change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as computer aided design where it is crucial to maintain the actual sizes of objects and angles between them as they're projected.

Void glOrtho(left, right, bottom, top, near, far) command creates a matrix for

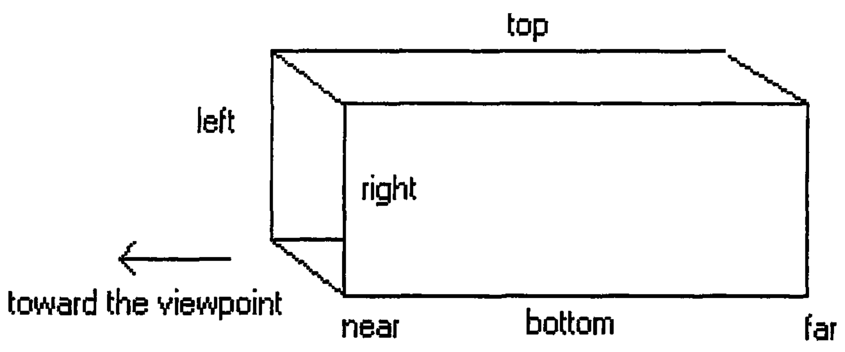


Figure 3.13 Orthographic Viewing Volume.

an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, near) and (right, top, near) are points on the near clipping plane that are mapped to the lower left and upper right corners of the viewport window, respectively.

For the special case of projecting a two dimensional image onto a two dimensional screen, use the Utility Library routine gluOrtho2D(). This routine is identical to the three dimensional version, glOrtho(), except that all the z coordinates for objects in the scene are assumed to lie between -1.0 and 1.0. If you are drawing two dimensional objects using the two dimensional vertex commands, all the z coordinates are zero; thus, none of the objects are clipped because of their z values.

Void glOrtho2D(left, right, bottom, top); creates a matrix for projecting two dimensional coordinates onto the screen and multiplies the current projection matrix by it. The clipping region is a rectangle with the lower left corner at (left, bottom) and the upper right corner at (right, top).

3.8 Viewport Transformation

3.8.1 Viewing Volume Clipping

After the vertices of the objects in the scene have been transformed by the modelview and projection matrix, any primitives that lie outside the viewing volume are clipped.

The viewport is the rectangular region of the window where the image is drawn. The viewport is measured in window coordinates, which reflect the position of pixels on the screen relative to the lower left corner of the window. Keep in mind that all vertices have been transformed by the modelview and projection matrix by this point, and vertices outside the viewing volume have been clipped.

3.8.2 Defining the Viewport

By default the viewport is set to the entire pixel rectangle of the window that is opened. You use the glViewport() command to choose a smaller drawing region; for example, you can subdivide the window to create a split screen effect for multiple views in the same window.

Void glViewport(x, y, width, height); command defines a pixel rectangle in the window into which the final image is mapped. The (x, y) parameter specifies the lower left corner of the viewport, and width and height are the size of the viewport rectangle. By default the initial values are (0, 0, winWidth, winHeight). The aspect ratio of a viewport is equal to the aspect ratio of the viewing volume. If the ratios are different, the projected image will be distorted when mapped to the viewport. Note that subsequent changes to the size of the window don't affect the viewport. The application should detect window resize events and modify the viewport appropriately.

4. BASIC IMAGE PROCESSING TECHNIQUES

4.1 Introduction to Image Processing

Image processing algorithms to manipulate the images mathematically involve integrals, however the necessity of sampling the images for computer use simplifies these formulae in terms of replacing integration operations by summation operations. Manipulation of the image can be of many forms. Briefly, the main goals in classical image processing are:

1) Image enhancement, to improve the appearance or highlight certain details of an image,
2) Image segmentation, the categorization or classification of elements or structures within an image,
3) Image manipulation, a general term including translation, scaling, rotation.

4.2 Point Processes

Point processes are algorithms to modify images on a pixel by pixel basis. Each pixel is replaced by a new pixel whose value depends only on the current pixel's value, and/or its location. These processes require only a single pass over the image data. Some of the examples for point processing are the image negation, thresholding, brightness adjustment and histogram equalisation.

4.2.1 Image Negation

Image negation is to make an image resemble a photographic negative. All the light regions of an image should be made dark, and all the dark regions should be made light. In practise, this can be achieved by stepping through the image pixel by pixel, subtracting the current pixel value from the maximum possible value. The result of this operation becomes the new pixel value. Alternatively, image negation can be achieved with the use of a lookup table. For a 256 level greyscale image, the first entry in the lookup table would contain the pixel value 255, and the last entry would contain the value 0. The lookup table is then applied to the image, thus negating it.

4.2.2 Thresholding

This technique is based upon a simple concept. If the grey−level interval is 0−65500, a parameter called the brightness threshold θ is chosen and applied to the image a[x,y] as follows:

if $a[x, y] < \theta, \quad a[x, y] = 0,$

else $a[x, y] = 65500.$

4.2.3 Brightness Adjustment

The object of brightness adjustment is to make the image lighter or darker by some factor. This can be done by stepping the image pixel by pixel, incrementing the pixel value by a positive constant to make the image lighter, or by a negative constant to make it darker. A lookup table can also be used for this purpose.

4.2.4 Histogram Equalization

An image histogram is a graph of pixel intensity distribution, showing the dynamic range of pixel values for that image. Specifically, it plots the number of occurences of each pixel value in the image, against pixel value.

The image histogram can also be thought of as a chart of the likelihood of a certain pixel value occuring at any point in the image, which we will call a probability table. In practise, to convert number of occurences to probability of occurence, the image histogram data should be normalised between 0 and 1. This is achieved by dividing the number of occurences of each pixel value by the total number of pixels in the image (65536 pixels for a 256 by 256 image). The result of this is that the sum of all the histogram data adds up to 1.

The next step is to create a lookup table where the n th value in the table is equal to the sum of the first n values in the probability table. This is sometimes called the cumulative histogram. Finally each item in the lookup table should be normalised between the minimum and the maximum pixel values, and rounded to the nearest integer.

4.3 Area Processes

Area processes transform an image, on a pixel by pixel basis, giving each pixel a new value, depending on the values, and/or positions, of its neighbours. The neighbourhood of a pixel forms a N by N grid around the pixel, where N is an odd number. An area process requires only a single pass over the image data, but it is computationally expensive than point processes, as the number of pixels read to calculate each new pixel value is N^2 greater than equilavent point process. Spatial filtering is used in this study for specific applications such as neighbourhood averaging, edge enhancement, low pass filtering.

4.3.1 Neighbourhood Averaging

Neighbourhood averaging is a smoothing process, performed by setting the value of each pixel in an image to the arithmetic mean of all the pixels in its neighbourhood. If the pixel neighbourhood is of size N by M, a convolution kernel of NxM weights is used, each set to 1, and the sum is divided by a scale factor of NxM. Repeated application of the neighbourhood averaging filter causes a greater degree of smoothing.

Below is the neighbourhood averaging kernel used in this study:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

4.3.2 Low Pass Filtering

Low pass filters attenuate the high frequency content of an image, whilst low frequencies are passed. This can help to remove the noise from an image, or allow the low frequency portion of an image to be examined more closely. Below is the low pass filter kernel used in this study.

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$

4.3.3 Laplacian Edge Enhancement

Laplacian edge enhancement uses an approximation of the Laplacian operator used throughout engineering and mathematics. It highlights the edges in all directions, and gives sharper edge definition than most other edge enhancement algorithms.

The Laplacian of a function f(x, y) is:

$$L(f(x,y) = d^2f/dx^2 + d^2f/dy^2 \tag{4.1}$$

where d^2f/dx^2 is the second partial derivative of f with respect to x, and d^2f/dy^2 is the second partial derivative of f with respect to y. For discrete functions, the second partial derivatives can be approximated by,

$$d^2f/dx^2 = f(x+1) - 2f(x) + f(x-1) \tag{4.2}$$

and

$$d^2f/dy^2 = f(y+1) - 2f(y) + f(y-1) \tag{4.3}$$

The Laplacian can therefore be approximated by,

$$L(f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \tag{4.4}$$

Convolution kernel representing the Laplacian transform is shown below.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

4.4 2-D Segmentation

One of the early tasks in image analysis is to segment an image into its constituent parts. By the segmentation process some portion of the image is extracted and accepted as an object. Image segmentation is an essential step in medical imaging applications. Especially in medical imaging, accurate segmentation is very crucial. In this study, Seeded Region Growing Algorithm (SRGA) [6] nbox for olcayki

4.4.1 Seeded Region Growing Algorithm (SRGA)

Since some commercial packages prefer using SRG algorithm for segmenting biomedical images, in this study, SRG algorithm has been implemented. The algorithm is fast, robust and parameter free. It takes an intensity image and a set of seeds as inputs. The seeds are represented as individual points. In the application, the user mark the seed or seeds to each object to be segmented. The SRG algorithm operates on the premise that the pixels within a region are similar. The algorithm grows the seed regions in an iterative fashion.At each iteration all those pixels that border the growing regions are examined. The pixel that is most similar to a region is appended to that region.

<u>4.4.1.1 Algorithm Description.</u> The SRG approach to image segmentation is to segment an image into regions with respect to a set of n seed regions. Each seed region is a connected component comprising one or more points and is represented by

a set A_j, where $j = 1, 2, ..., n$. Let T be the set of all unallocated pixels that border at least one of the A_j, i.e., $T = x \notin \cup A_j : N(x) \cap \cup A_j \neq \emptyset$ where N(x) represents the set of immediate neighbours (8-connectivity) of the pixel x. A single step of the algorithm involves examining the neighbours of each $x \in T$ in turn. If N(x) intersects a region Aj then a measure, $\delta(x)$, of the difference (similarity) between x and the intersected region is calculated. In the simplest case $\delta(x)$ is defined: $\delta(x) =\mid g(x) - mean_{y \in Aj} g(y) \mid$ where g(x) is the intensity grey value of the pixel x. If N(x) intersects more than one region then Aj is taken to be that region for which $\delta(x)$ is minimum. In this way a $\delta(x)$ value is determined for each $x \in T$. Finally the pixel $z \in T$ that satisfies $\delta(z) = min_{x \in T} \delta(x)$ is appended to the region. This process continues till all the image pixels have been assimilated. In implementing the SRG algorithm Adams and Bischof utilise a data structure called the sequentially sorted list (SSL). According to them, SSL is a linked list of pixel addresses, ordered with respect to $\delta(x)$. A pixel can be arbitrarily inserted into the list in the position prescribed by its $\delta(x)$ value. However only the pixel with the smallest $\delta(x)$ value can be removed from the SSL. Effectively, the SSL stores the points of the set T ordered according to $\delta(x)$. Adams and Bischof noted that their implementation does not update the previous entries in the SSL to reflect new differences from a region whose mean has been updated. They stated that "this leads to negligible difference in the results, but greatly enhanced speed" [6].

4.5 3-D Segmentation

3-D segmentation is based on 2-D segmentation process in this study. After the user represens the seeds interactively on the central slice, the toolbox calculates the seed intensity averages and then it calls the auto_ seeding function which is responsible for representing the seeds automatically for the rest of the dataset. The toolbox then asks for the slice interval to be segmented. The resulted files are kept in the pre-defined directory and region view utility is used for viewing all the segmented regions in 3-D manner. This function also support zooming and rotation utilities.

5. WINDOWS PROGRAMMING and OBJECT ORIENTED SOFTWARE IMPLEMENTATION

This chapter explains the basic principles of Windows Programming for the design of GUI (Graphical User Interface) using the Borland ObjectWindows C++ programming language.

5.1 Windows Programming Principles

This section describes the ObjectWindows 2.0 classes, explains how each class fits together with the others, and how to use each class.

5.1.1 Working with Class Hierarchies

- What can we do with a class,
- Inheriting members,
- Types of member functions.

5.1.1.1 Using a class. There are three basic things we can do with a class:

- Deriving a new class from it,
- Adding its behaviour to that of another class,
- Creation an instance of it.

```
class TNewWindow : public Twindow

{

        public:

        TNewWindow(...);

        .....

        .....

}
```

In order to use a class, we must create an instance of it.**Constructors** are being used for this purpose. The constructors call the base class' constructors and initialize any needed data members.

5.1.1.2 Inheritance. With ObjectWindows, we can derive new classes that inherit the behaviour of more than one class. Such "mixed" behaviour is different from the behaviour you get from a single inheritance derivation. Instead of inheriting the behaviour of the base class, we are inheriting and combining the behaviour of several classes.

TInputDialog inherits all the data members of TDialog and TWindow and adds the data members it needs to be an input dialog box. TInputDialog makes it possible to store and retrieve user input data.

5.1.1.3 Member Functions. There are four types of ObjectWindows member functions:

- Virtual,
- Pure Virtual,
- Nonvirtual,
- Default Placeholder.

Virtual functions can be overridden in derived classes. They differ from Pure Virtual functions in that they don't have to be overridden in order to use the class. Pure Virtual functions must be overridden in derived classes. Nonvirtual and Default placeholder functions can also be overridden in the derived class. For example, TWindow::SetCaption is nonvirtual since we don't need to change the way a window's caption is set. TWindow::EvLButtonClk function is an example for a Default placeholder function and it performs the default message processing. It can be overridden in the generated new class.

5.1.2 Object Typology

The ObjectWindows hierarchy has many different types of classes that you can use, modify or add to. We can seperate what each class does into the following groups:

- Windows,
- Dialog boxes,
- Controls,
- Graphics,
- Printing,
- Modules and applications,
- Doc and View applications,

- Miscellaneous Windows elements.

5.1.2.1 Window Classes. An important part of any Windows applicaton is, of course, the window. ObjectWindows provides several different window classes for different types of window.

- Windows : TWindow,
- Frame windows : TFrameWindow,
- MDI windows : TMDIClient, TMDIChild, TMDIFrame,
- Decorated windows : TDecoratedMDIFrame, TDecoratedFrame.

5.1.2.2 Dialog Boxes Classes.

- Common Dialog boxes : TFileOpenDialog and TFileSaveDialog, TChooseFont-Dialog, TPrintDialog, TFindDialog, TReplaceDialog,
- Other dialog boxes : TInputDialog.

5.1.2.3 Control Classes.

- Standard Windows controls : List boxes, scroll bars, buttons, radio buttons, group boxes, edit controls, static controls and combo boxes,
- Widgets : THSlider, TVSlider,
- Gadgets,
- Decorations : TMessageBar, TControlBar, TToolBox, TStatusBar.

5.1.2.4 Graphics Classes.

- DC classes : TDC,
- GDI objects : TGDI Object.

5.1.2.5 Printing Classes.

- Printing classes : TPrinter and TPrintout.

5.1.2.6 Modules and Applications Classes.

- Modules and Applications classes : TApplication, TModule.

5.1.2.7 Doc and View Classes.

- Documentation and view classes : TDocManager, TDocument, Tview.

5.1.2.8 Miscellaneous Classes.

- Menus : TMenu,
- Clipboard : TClipboard.

5.2 Object Oriented Software Implementation

5.2.1 New Application Class Declaration

In the study, simple application code shown below was realized. Since the application uses the TApplication and TFrameWindow Object Window classes, the source file must include applicat.h and framewin.h header files.

The class Application is derived from the ObjectWindows TApplication class. Every ObjectWindows application has a TApplication object known as the application object (Application). Overriding TApplication gives you access to the workings of the application object and lets you override the necessary functions to make the application work the way you want. The code shown below shows the Aplication class used in the study.

```
class Application : public TApplication

{

    public:

    Application();

    void InitMainWindow();

};

Application::Application() : TApplication() { }
```

5.2.2 OwlMain and Run Function

In addition to an application object, every ObjectWindows application has an OwlMain function. The application object is actually created in the OwlMain function with a simple declaration. OwlMain is the ObjectWindows equivalent of the WinMain function in a regular Windows application. You can use OwlMain to check command-line arguments (not used in 3DVIEW) before the application begins execution.

```
int OwlMain(int /*argc*/, char* /*argv*/[])

{ return Application().Run(); }
```

To start execution of the application, we first call the application object's Run function. The Run function first calls the InitApplication function. After the InitApplication function returns, Run calls the InitInstance function, which initialize each instance of the application. The default InitInstance function calls the function InitMainWindow, which initializes the application's main window then creates and displays the main window. Application function overrides the InitMainWindow function. This makes it possible to design the main window however you want it. The SetMainWindow function sets the application's main window to a TFrameWindow or TFrameWindow derived object passed to the function and the title is "VIEW". GetMainWindow function makes it possible to add the MENU and window icon to the main window. MENU and ICON resources was constructed by using BRW utility. Whole code written for all the things mentioned above is shown below.

```
void Application::InitMainWindow()

{

    TApplication::InitMainWindow();
```

```
          SetMainWindow(new DecFrame (0,”VIEW”,new TGLWindow, true));

          GetMainWindow()- >AssignMenu(MENU_1);

          GetMainWindow()- >SetIcon(this, ICON_1);

     }
```

5.2.3 New Windows Class Declaration

The class TGLWindow is derived from Twindow class and it looks like this:

```
     class TGLWindow : public TWindow

     {

          HGLRC hRC;

          HDC hDC;

          public:

          TGLWindow(TWindow* = 0,const char far* = 0, TModule* = 0);

          ~TGLWindow();

          void AssignMenu();

          void SetupWindow();

          bool bSetupPixelFormat(HDC);
```

```
        void draw();

        void init();

        void EvPaint();

        void EvSize(uint,TSize&);

        void EvLButtonDown(uint, TPoint&);

        void EvRButtonDown(uint, TPoint&);

        DECLARE_ RESPONSE_ TABLE(TGLWindow);

    };
```

The response table definition sets up your class to handle Windows events and to pass each event on the proper event-handling function. As a general rule, event-handling functions should be protected. This prevents classes and functions outside your own class from calling them. The response table declaration should take its place in the class declaration.

```
    class DecFrame : public TDecoratedFrame

    {

        public:

        DecFrame(TWindow*,const char far*,TWindow*, bool track);

        private:
```

```
        TDirPath Dirpath;

        DECLARE_ RESPONSE_ TABLE(DecFrame);

        void cm_ fileopen();

        void cm_ filesave();

        void cm_ show_ hist();

        void cm_ smoothing();

        void cm_ 3d_ segmentation();

        void cm_ 2d_ segmentation();

        void cm_ 3d_ region_ view();

        void cm_ about();

        void cm_ help();

        .     .     .

    };
```

SetMainWindow(new DecFrame (0,"VIEW",new TGLWindow, true)); command in the InitMainWindow function binds the application with a new window with title "VIEW", and with a decorated frame.

5.2.4 Response Table Declaration

The first line of a response table definition is always the DEFINE_ RESPONSE _ TABLEX macro. The value of X depends on your class' inheritance, and is based on the number of immediate base classes your class has. The last line of a response table definition is always the END_ RESPONSE_ TABLE macro, which end the event response table definition. Between the DEFINE_ RESPONSE_ TABLEX macro and the END_ RESPONSE_ TABLE macro other macros with their handling functions take place.

Response Table for the TGLWindow

```
DEFINE_ RESPONSE_ TABLE1(TGLWindow, TWindow)

    EV_ WM_ PAINT,

    EV_ WM_ SIZE,

    EV_ WM_ LBUTTONDOWN,

    EV_ WM_ RBUTTONDOWN,

END_ RESPONSE_ TABLE;
```

Response Table for DecFrame

```
DEFINE_ RESPONSE_ TABLE1(DecFrame,TDecoratedFrame)

    EV_ COMMAND(CM_ FILEOPEN,cm_ fileopen),

    EV_ COMMAND(CM_ FILESAVE,cm_ filesave),
        EV_ COMMAND(CM_ SHOW_ HIST,cm_ show_ hist),

        EV_ COMMAND(CM_ SMOOTHING,cm_ smoothing),

        EV_ COMMAND(CM_ 3D_ SEGMENTATION,cm_ 3d_ segmentation),

...

    END_ RESPONSE_ TABLE;
```

The parameters that are passed to the EV_ COMMAND macro:

1) The enum value of the event is defined in BRW while constructing the MENU resource. These values are stored in one of the include header files and shown below.
2) The event-handling function.

As it can be seen, four protected functions in the TGLWindow are EvLButton-Down, EvLButtonDown, EvPaint and EvSize. When TGLWindow receives a WM_ LBUTTONDOWN event, it passes it to the appropriate function.

5.2.5 Adding a Menu Resource to the Main Window

Here is the definition of Event Identifiers. These identifiers were constituted by BRW utility automatically in the trial.rc. All the Menu items such as Open File, Save File, 2-D Segmentation etc. were designed by BRW utility.

```
define CM_ FILEOPEN 1008

define CM_ FILESAVE 1009
define CM_ SHOW_ HIST 1006

define CM_ SMOOTHING 1007

define CM_ ABOUT 1000

define CM_ 2D_ SEGMENTATION 106

...
```

The menu resource is attached to application in the InitMainWindow function. This is done by calling the main window's AssignMenu function shown below. In order to get the main window GetMainWindow function is called. In addition, ICON_ 1 resource that is constituted by BRW is added to the main window by the SetIcon function. This icon takes its place at the left up corner of the window.

```
void Application::InitMainWindow()

{;

    TApplication::InitMainWindow();

    SetMainWindow(new DecFrame (0,"VIEW",new TGLWindow,true));

    GetMainWindow()− >AssignMenu(MENU_ 1);

    GetMainWindow()− >SetIcon(this, ICON_ 1);

};
```

5.2.6 Changing The Current Menu Resource

Whenever the user attempts to add a resource to the toolbox, it is necessary to request a new resource from BRW as it is shown in Figure 5.1. Then by using BRW tools the resource is designed.

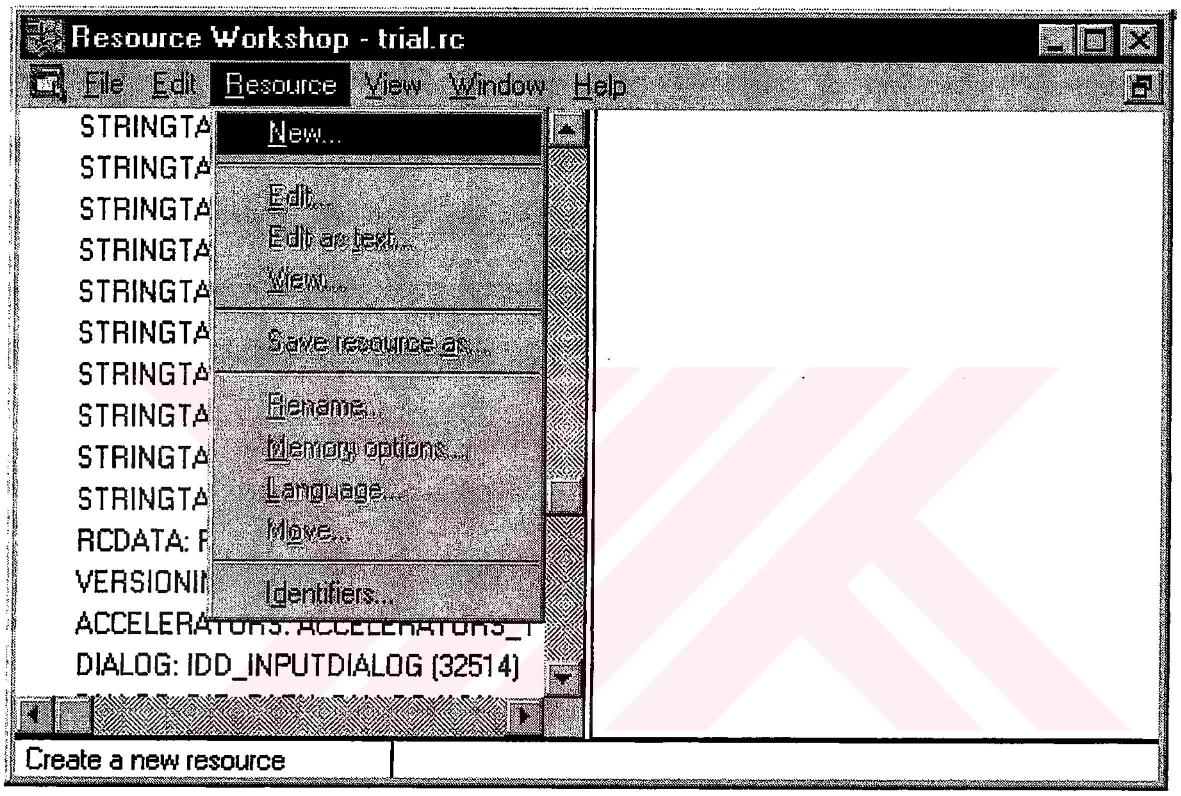


Figure 5.1 BRW, requesting a new resource.

In order to change the MENU resource completely by adding a new MENU resource to the toolbox, a new menu resource is requested from BRW as it is shown in Figure 5.2. When the user completes the MENU design, in the software, the AssignMenu function parameter which is an identifier for the MENU resource, should be updated by the new MENU resource identifier. Then in the InitMainWindow function new MENU resource will be automatically inserted to the window.

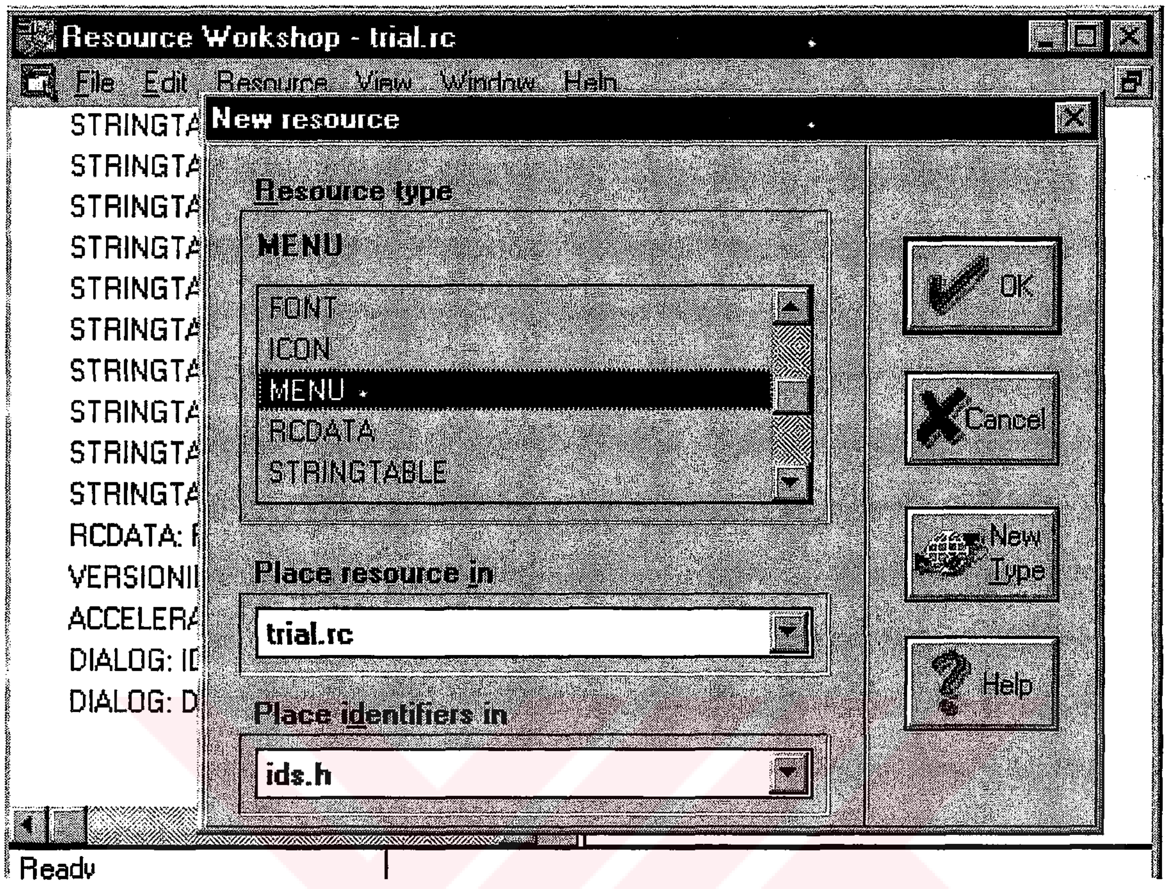


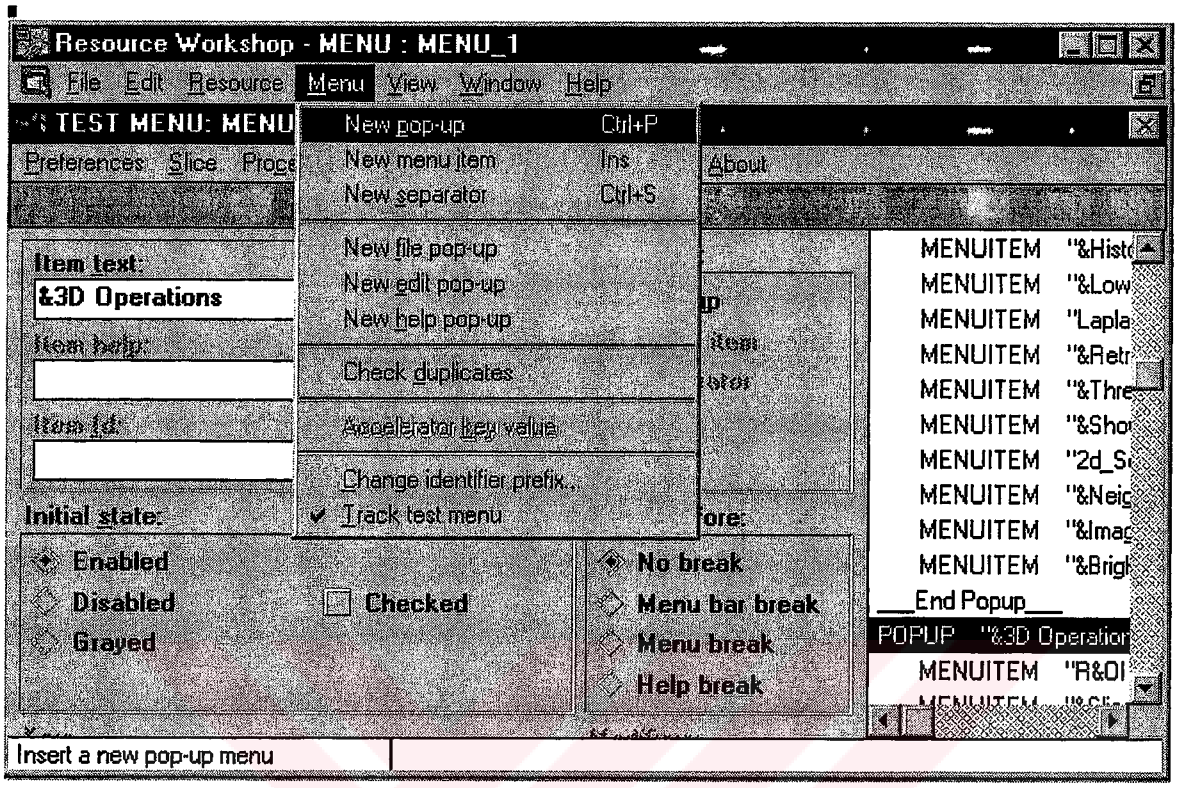
Figure 5.2 BRW, creating a new MENU resource..

5.2.7 Adding New Functions for New Algorithms

The user may specify a new function or functions for specific algorithms and then may want to see the results by using the viewing part of the toolbox. For this purpose, A new pop-up menu or a new menu-item resource should be designed and inserted to the toolbox. And also some addings should be done to the software. Below, the necessary steps that should be taken for this purpose are given in detail.

5.2.7.1 Adding a New Pop-up Menu to the Menu Resource.

The user may want to attach a new pop-up menu to the current MENU resource for implementing some algorithms on the image dataset. This is done by using the New Pop-up Menu command in the BRW utility as it is shown in Figure 5.3.



Item text: &3D Operations

Initial state: Enabled

No break

✓ Track test menu

```
MENUITEM "&Histo
MENUITEM "&Low
MENUITEM "Lapla
MENUITEM "&Retr
MENUITEM "&Thre
MENUITEM "&Sho
MENUITEM "2d_S
MENUITEM "&Neig
MENUITEM "&Imag
MENUITEM "&Brigh
___End Popup___
POPUP "&3D Operation
MENUITEM "R&OI
```

Insert a new pop-up menu

Figure 5.3 BRW, inserting a new Pop-up Menu Resource to the toolbox.

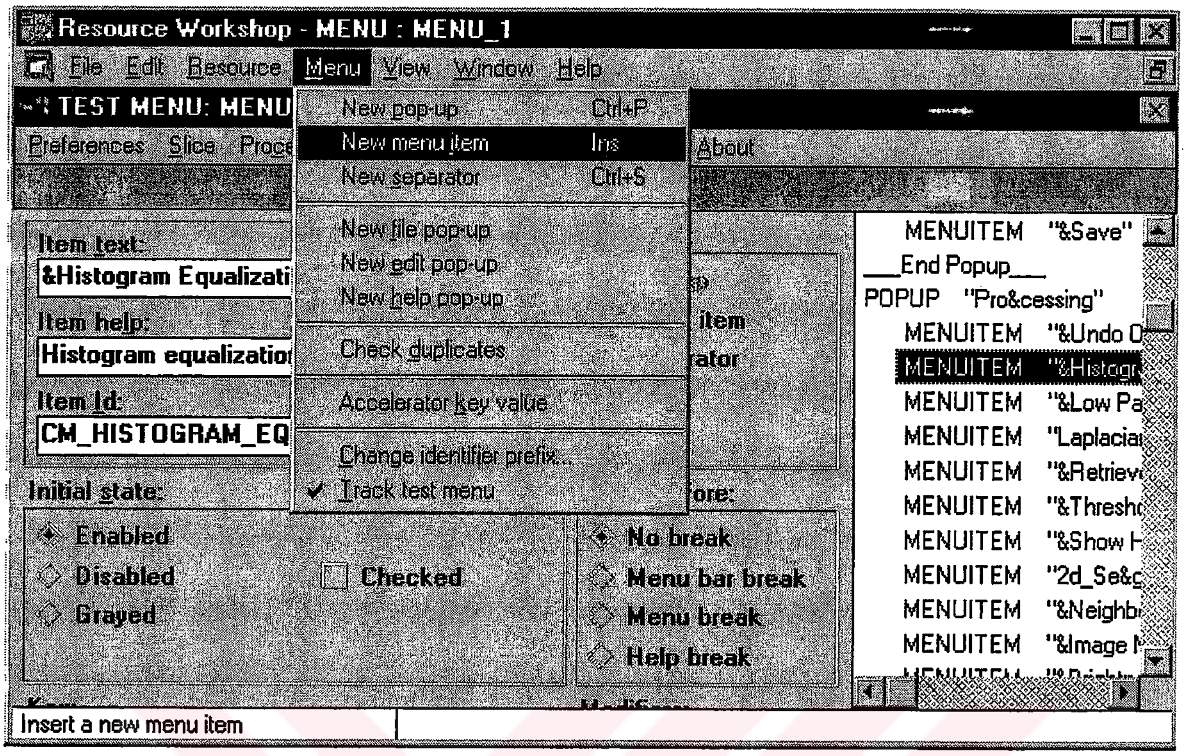
<u>5.2.7.2 Adding a New Menuitem to the Menu Resource.</u> Now, the next step is to fill the pop-up menu list with the menu-items that will realize specific algorithms for the biomedical image dataset. This is shown in Figure 5.4.

As it is mentioned before, we use the RESPONSE_ TABLE to handle the event handling functions and the parameters that are passed to the EV_ COMMAND macro:

1) The enumarated resource value of the event that is defined in BRW. These values are stored in one of the include header files.

2) The event-handling function.

In the software, new menu-items' resource identifiers and their event handling



Item text: &Histogram Equalizati

Item help: Histogram equalizatio

Item Id: CM_HISTOGRAM_EQ

```
    MENUITEM  "&Save"
___End Popup___
POPUP  "Pro&cessing"
    MENUITEM  "&Undo O
    MENUITEM  "&Histogr
    MENUITEM  "&Low Pa
    MENUITEM  "Laplacia
    MENUITEM  "&Retriev
    MENUITEM  "&Thresh
    MENUITEM  "&Show H
    MENUITEM  "2d_Se&g
    MENUITEM  "&Neighb
    MENUITEM  "&Image M
```

Insert a new menu item

Figure 5.4 BRW, inserting a new Menu-item resource to the toolbox.

function names (new algorithm function name) is written to window's RESPONSE_ TABLE. Then the function which will realize a specific algorithm is written and inserted to the software.

5.2.8 Adding Decorations to the Main Window

In order to create a decorated window including a control bar and button gadgets on it, these steps should be taken:

1) Changing the main window from a Twindow to a TdecoratedFrame.

2) Creating a control bar, along with its button gadgets, and inserting it into the decorated frame.

3) Adding resources constituted by BRW, such as bitmaps for the button gadgets.

Changing from a Twindow to a TDecoratedFrame is quite easy. For this purpose, Class DecFrame was constituted from TDecoratedFrame for overlapping and this is passed as a parameter to the SetMainWindow function as it is shown below.

```
void Application::InitMainWindow()

{;

    TApplication::InitMainWindow();

    SetMainWindow(new DecFrame (0,"VIEW",new TGLWindow,
true));

    GetMainWindow()– >AssignMenu(MENU_1);

    GetMainWindow()– >SetIcon(this, ICON_1);

};
```

5.2.9 Creating the Control Bar

Firstly, the actual TControlBar object is constructed. Then the gadgets that make up the controls on the bar are created and inserted into the constructed control bar.

Here is the control bar constructor:

```
TControlBar *cbar = new TControlBar(this);
```

Button Gadgets are used as control bar buttons. A button gadget is associated with a BITMAP resource and the BITMAP resource can be created in BRW, as shown in Figure 5.5 and Figure 5.6. Button gadgets associate a BITMAP button resource with an event identifier. When the user presses a button gadget, it sends that event identifier. You can set this up so that pressing a button on the control is just like making a choice from a menu. Button gadgets are created using the TButtonGadget class. The TButtonGadget constructor takes six parameters, of which we need to use only the first two:

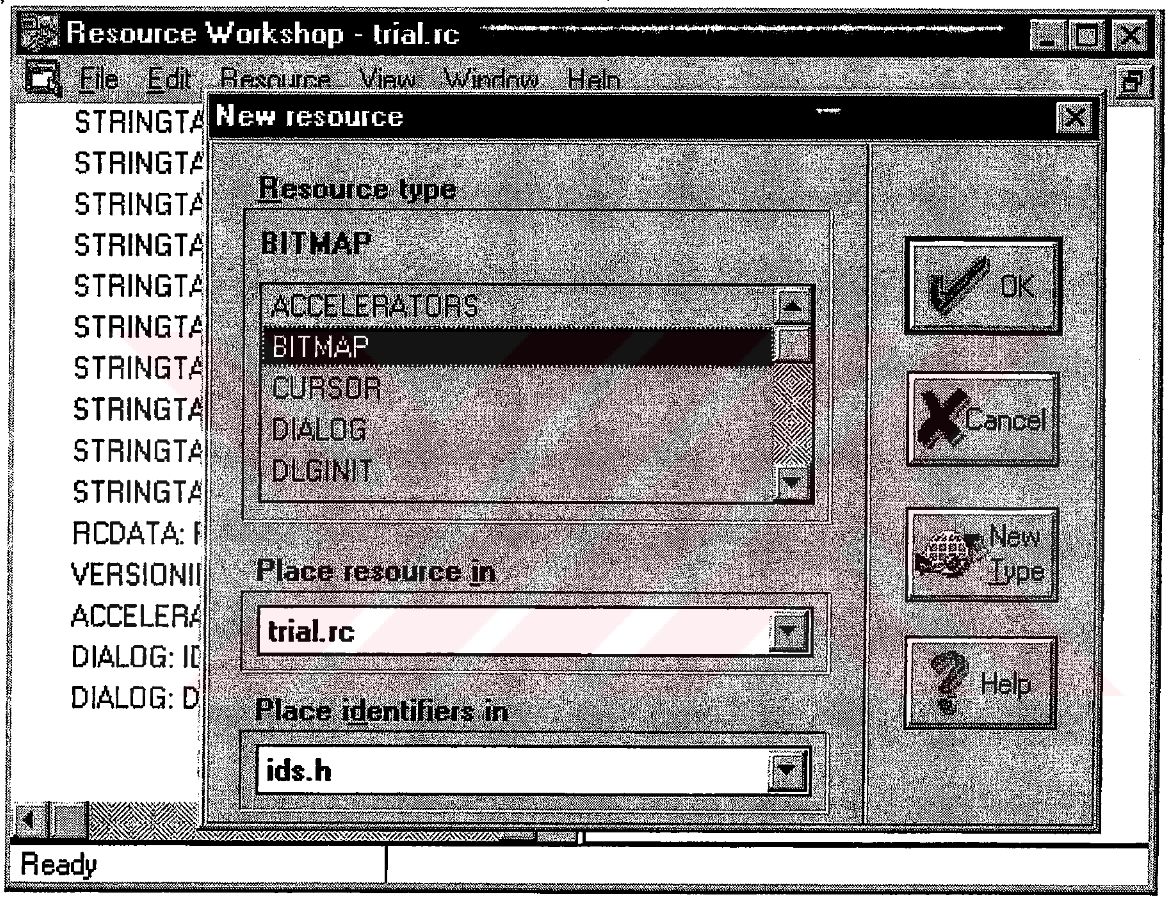


Figure 5.5 BRW, creating a new BITMAP resource for a button gadget.

The first parameter is a reference to a TResId object. This should be resource identifier of the bitmap. The second parameter is the gadget identifier for this button gadget. Once the gadgets are constructed, Insert function is used for inserting the button gadgets into the control bar. Some button gadgets used in the study are given below:

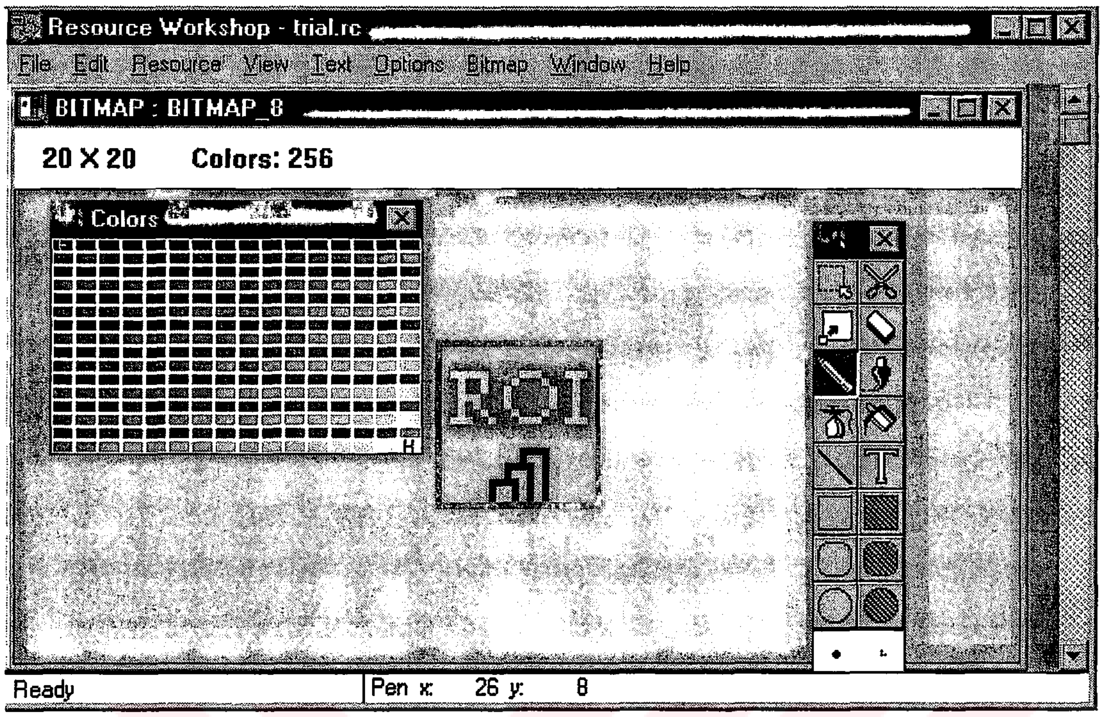

Resource Workshop - trial.rc
File Edit Resource View Text Options Bitmap Window Help
BITMAP : BITMAP_8
20 X 20 Colors: 256
Colors
ROI
Ready
Pen x: 26 y: 8

Figure 5.6 BRW, constructing a new BITMAP resource for a button gadget.

```
cbar– >Insert(*new TButtonGadget(BITMAP_3,CM_ZOOMPLUS));

cbar– >Insert(*new TButtonGadget(BITMAP_4,CM_ZOOMMINUS));

cbar– >Insert(*new TButtonGadget(BITMAP_7,CM_ABOUT));

...
```

For inserting a seperator gadget into the control bar the command shown below was used.

```
cbar– >Insert(*new TTextGadget(–1, TTextGadget::Left,0.1, ""));
```

Up to now, button gadgets were created and inserted to control bar. Now it is time to insert the control bar into the decorated window. Insert command shown below is used for that purpose.

```
        Insert(*cbar, TDecoratedFrame::Top);
```

5.2.10 New Dialog box class declaration

A new dialog class called TDirPathDlg is inherited from the base class, TDialog class. This dialog box will be used for setting up the directory paths that the toolbox needs to access. For this purpose, by using BRW utility, a new dialog box resource, DIR_ PATH_ INFO was created.

```
        class TDirPathDlg : public TDialog

        {

                public:

                TDirPathDlg(TWindow* parent, const char* name, TDirPath transfer);

                ~TDirPathDlg()delete edit;

                TEdit *edit;

        };
```

Edit box controls in which we define the directory paths, are constructed by using TEdit class object constructor shown below:

```
        TDirPathDlg::TDirPathDlg(TWindow* parent, const char* name, TDirPath
transfer) : TDialog(parent, name), TWindow(parent)

        {
```

```
        edit = new TEdit(this, 140, sizeof(transfer.roi_ ddp));

        edit = new TEdit(this, 141, sizeof(transfer.roi_ sdp));

        edit = new TEdit(this, 142, sizeof(transfer.ddp_ 2d_ seg));

        edit = new TEdit(this, 143, sizeof(transfer.sdp_ org_ dataset));

        edit = new TEdit(this, 144, sizeof(transfer.ddp_ 3d_ seg));

        edit = new TEdit(this, 145, sizeof(transfer.sdp_ 3d_ seg));

    };
```

The second parameter in the TEdit object constructor refers to the resource identifier of the designed dialog box, edit box objects. Now it is time to construct the dialog box by using the Dialog box constructor shown in the following function.

```
    void DecFrame::cm_ dir_ path_ definition()

    {

        FILE *fp;

        if (TDirPathDlg(this,"DIR_   PATH_   INFO", DirPath).Execute() ==
IDOK)

         {;

         fp=fopen("C:/DIRINFO.TXT","w+t");

         fwrite(DirPath.roi_ ddp,sizeof(DirPath.roi_ ddp),1,fp);
```

```
            fwrite(DirPath.roi_sdp,sizeof(DirPath.roi_sdp),1,fp);

            fwrite(DirPath.ddp_2d_seg,sizeof(DirPath.ddp_2d_seg),1,fp);

            fwrite(DirPath.sdp_org_dataset,sizeof(DirPath.sdp_org_dataset),1,fp);

            fwrite(DirPath.ddp_3d_seg,sizeof(DirPath.ddp_3d_seg),1,fp);

            fwrite(DirPath.sdp_3d_seg,sizeof(DirPath.sdp_3d_seg),1,fp);

            fclose(fp);

            strcpy(ddproi,DirPath.roi_ddp);

            strcpy(sdproi,DirPath.roi_sdp);

            strcpy(ddp2dseg,DirPath.ddp_2d_seg);

            strcpy(sdporiginaldset,DirPath.sdp_org_dataset);

            strcpy(ddp3dseg,DirPath.ddp_3d_seg);

            strcpy(sdp3dseg,DirPath.sdp_3d_seg);

      }

  };
```

The user edits the edit box in the constructed dialog box and then the represented directory paths are written to global variables as well as to the file called DIRINFO.txt. This file is used as a retrieve file whenever the user runs the application, the dialog box retrieves the previous saved directory paths from it.

5.2.10.1 Adding a New Dialog Box. By using the BRW, a new dialog box can be added to the toolbox. To do this, a new dialog box resource is requested from the BRW as it is shown in Figure 5.7.

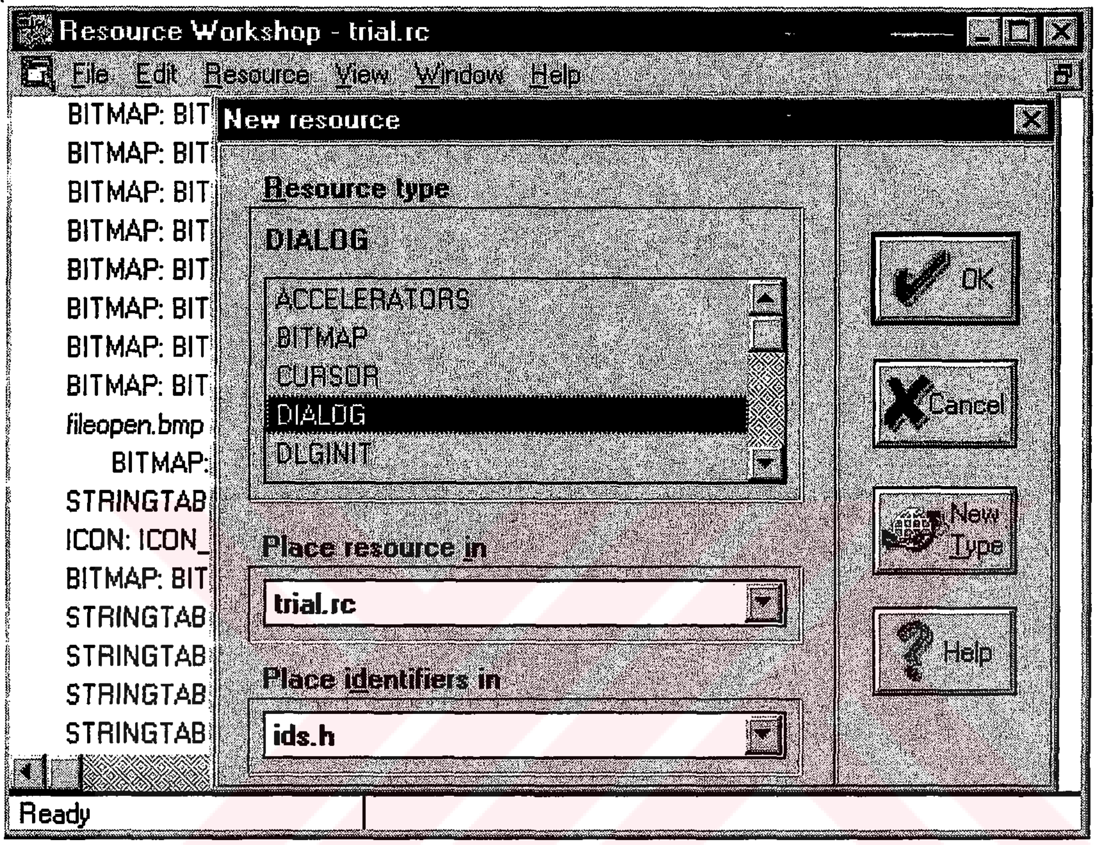
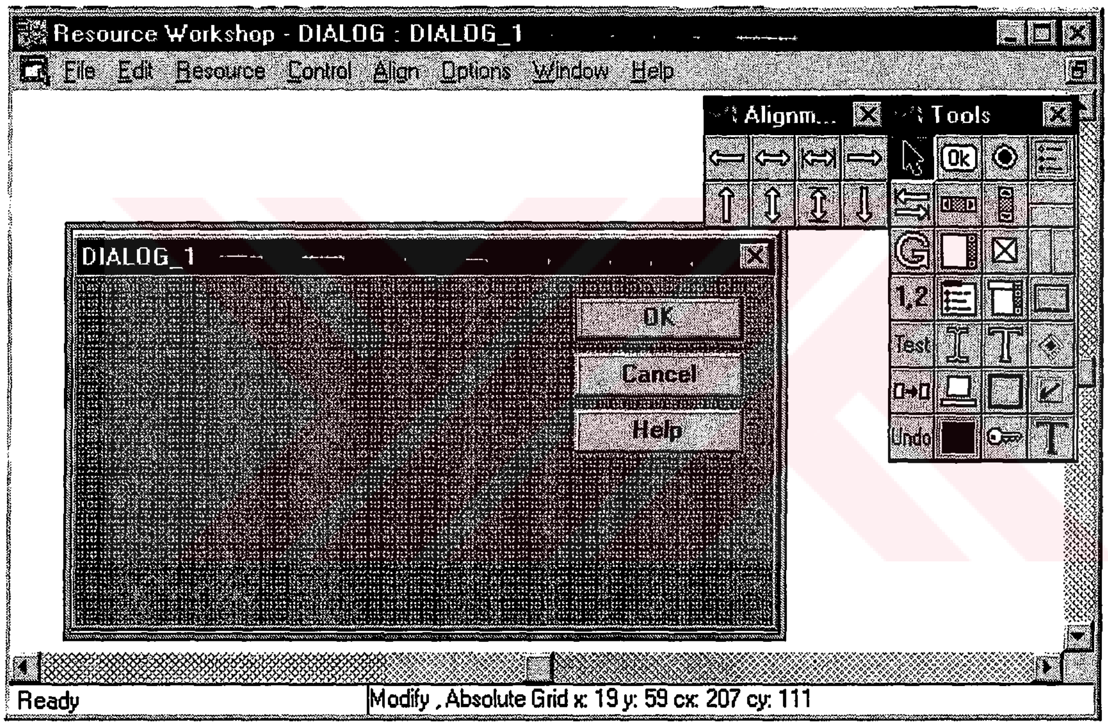


Figure 5.7 BRW, creating a new dialog box resource.

By using the BRW tools shown in Figure 5.8, the dialog box is constructed.


Resource Workshop - DIALOG : DIALOG_1
File Edit Resource Control Align Options Window Help
Alignm...
Tools
DIALOG_1
OK
Cancel
Help
Test
Undo
1,2

Ready
Modify , Absolute Grid x: 19 y: 59 cx: 207 cy: 111

Figure 5.8 BRW, constructing a new dialog box resource.

6. IMPLEMENTATION

6.1 Toolbox Construction

All the toolbox capability tests have been realized on Windows 95 and Windows 98 operating system platforms using a real human head dataset comprising 128 slices. In the dataset, each slice is represented by 256x256 pixels and each pixel has two bytes or one word in lenght.

As it is mentioned in Chapter 5, by using TWindow class, VIEW window was created and the toolbox icon was connected to it. And also a pop up menu resource has been attached to the main window frame. Both the toolbox icon and the menu resource were designed by using the BRW utility. The SetMainWindow function sets the application's main window to a TDecoratedFrameWindow and the title is "VIEW". GetMainWindow function was used in order to add the MENU and the toolbox icon to the main window. The functions used for this purpose are the AssignMenu and the SetIcon function called from the window's InitMainWindow function. TDecoratedFrame has been created from TWindow and a control bar was inserted to it. All the menu-item shortcuts designed as a bitmap resource and they all have been installed into the contol bar. Firstly, the actual TControlBar object is constructed. Then the gadgets that make up the controls on the bar are created and inserted into the constructed control bar.

The VIEWPORT area represents the work or graphics area that the toolbox can use for drawing purposes. The software code result is shown in Figure 6.1. In addition, the status bar was generated so that the toolbox gives an information whenever the user strolls through the menu shortcuts (bitmaps) or menu-items. The strings for the status bar has been represented in BRW.

Many applications work with pre–defined directories. 3DVIEW is one of them.

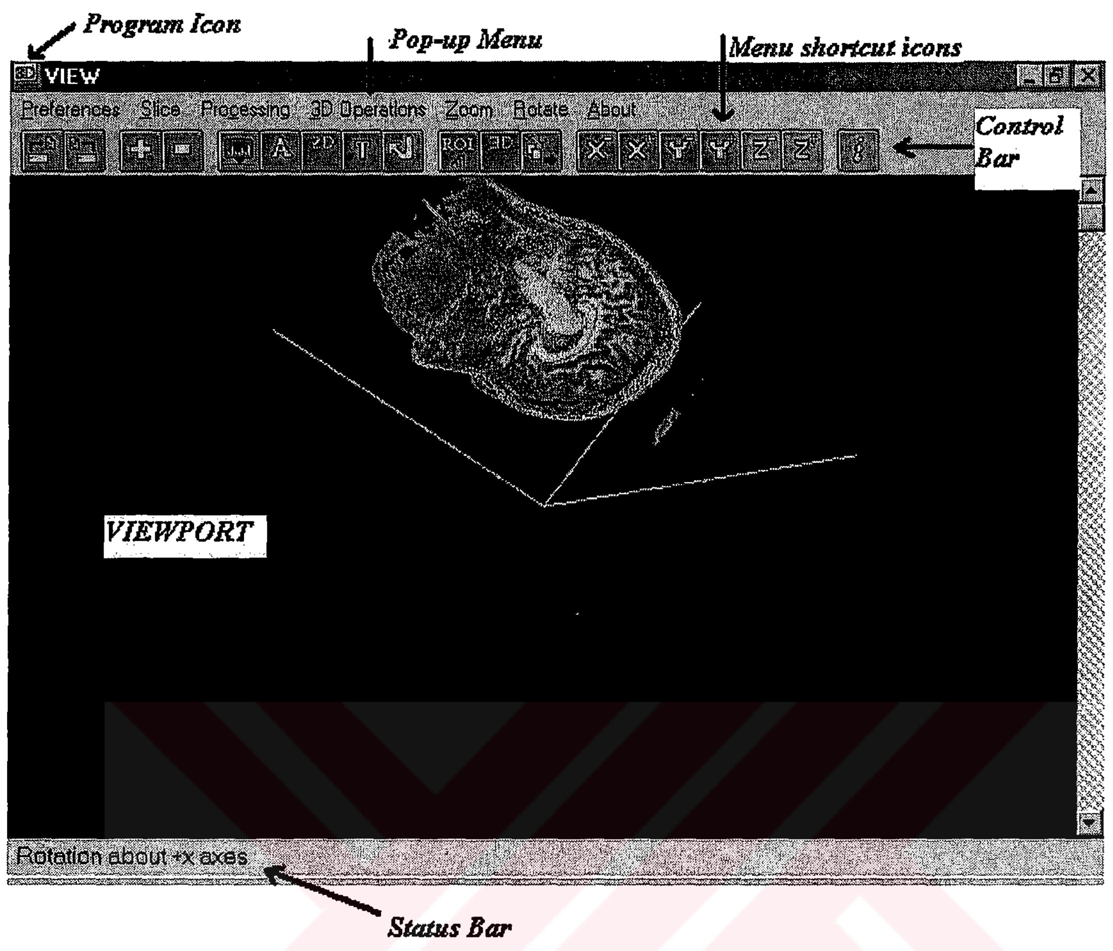
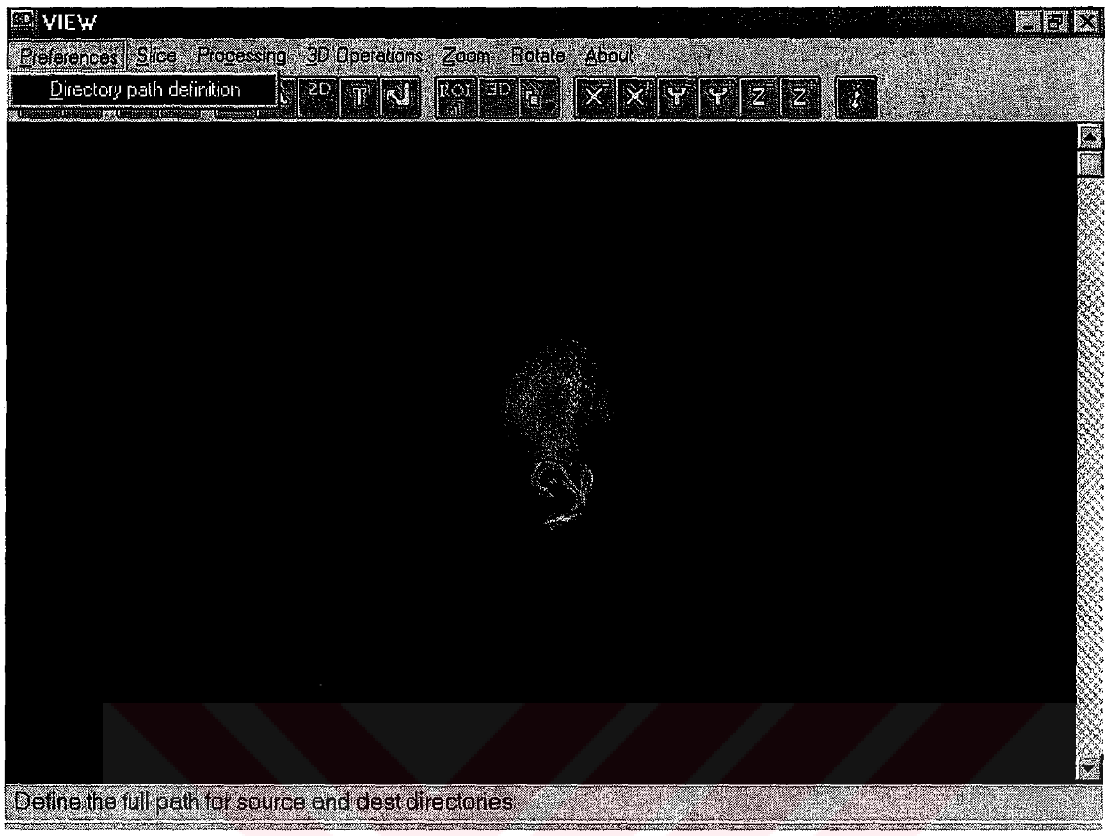


Figure 6.1 Constituted Window by using C++ Object Windows 2.0.

The Preferences popup menu is shown in Figure 6.2. By using the dialog box, full directory paths for the input and the output files can be specified.

In order to create a new dialog box for directory path definitions, a new dialog box resource has been requested from BRW and by using BRW tools, requested dialog box shown in Figure 6.3 has been designed. The user edits the edit box in the constructed dialog box shown in Figure 6.3. And then the represented directory paths are written to global variables as well as to the file DIRINFO.txt. This file is used as a retrieve file, whenever the user open the application, the dialog box retrieves the saved directory paths from it.

In Figure 6.4, Slice pop up menu is shown. In order to view the biomedical



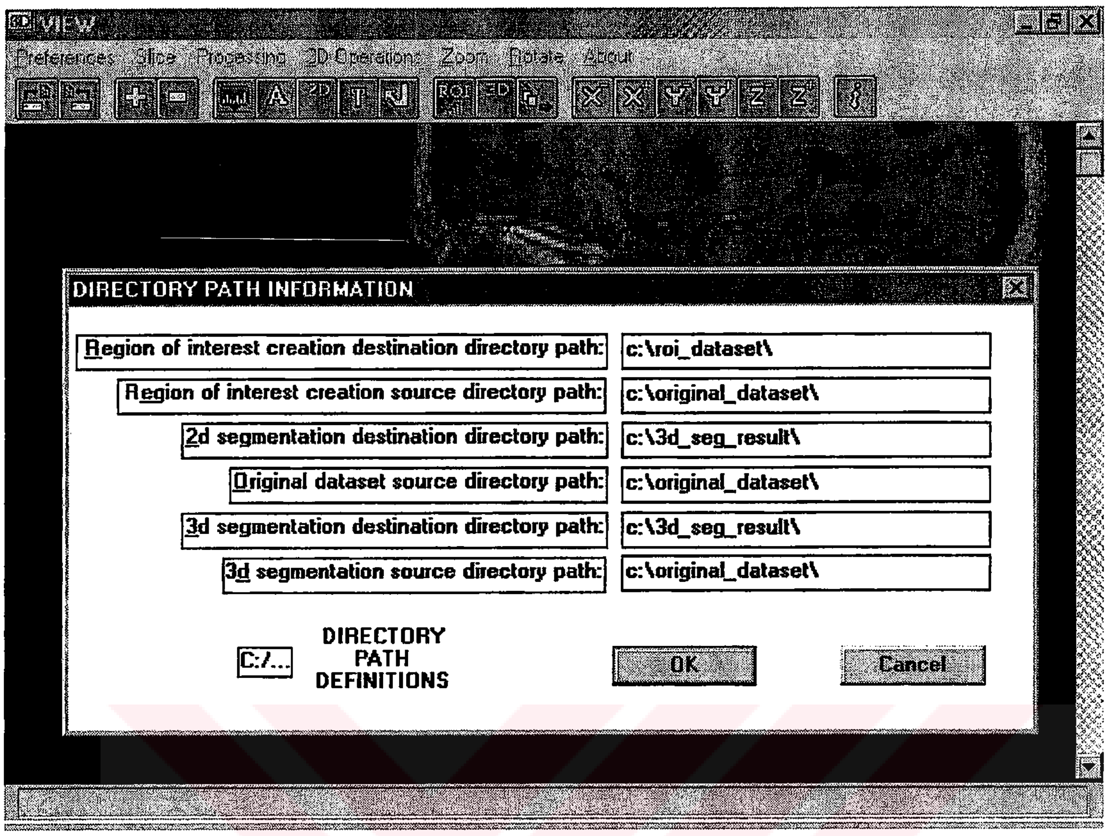
Define the full path for source and dest directories

Figure 6.2 Preferences, Directory Path Definitions menu-item.

images slice by slice, Open Slice function was written. And also in order to save the modifed image to disk, Save Slice function has been added to pop up menu.

Open Slice and Save Slice dialog box object has been created from Open File Common Dialog Box class. Whenever the user attempts to open or to save a slice, the dialog box showing the directories and files on the current disk appears on the screen. The user selects a file interactively to work with it.

Open Slice dialog box and Save Slice dialog box are shown in Figure 6.5 and 6.6 respectively.



DIRECTORY PATH INFORMATION

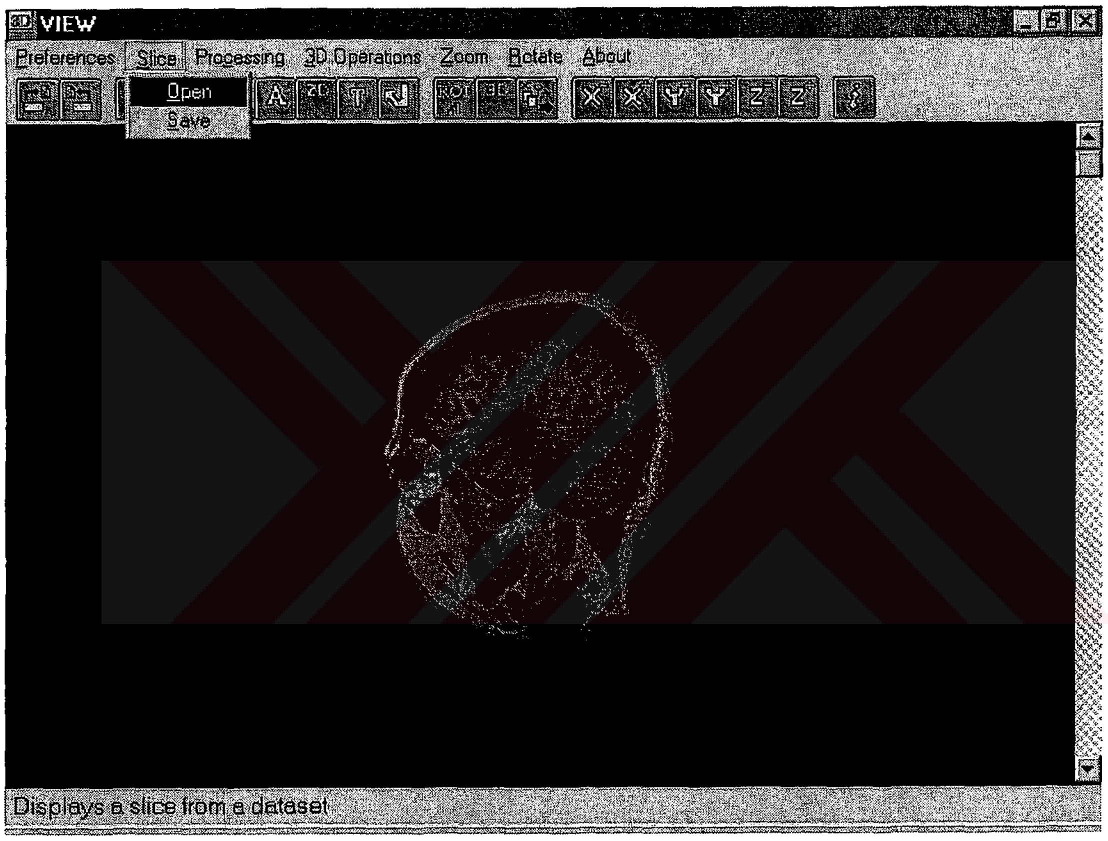
Region of interest creation destination directory path:	c:\roi_dataset\
Region of interest creation source directory path:	c:\original_dataset\
2d segmentation destination directory path:	c:\3d_seg_result\
Original dataset source directory path:	c:\original_dataset\
3d segmentation destination directory path:	c:\3d_seg_result\
3d segmentation source directory path:	c:\original_dataset\

Figure 6.3 Preferences, Directory Path Definitions Dialog Box.

6.2 Inserting New Algorithms

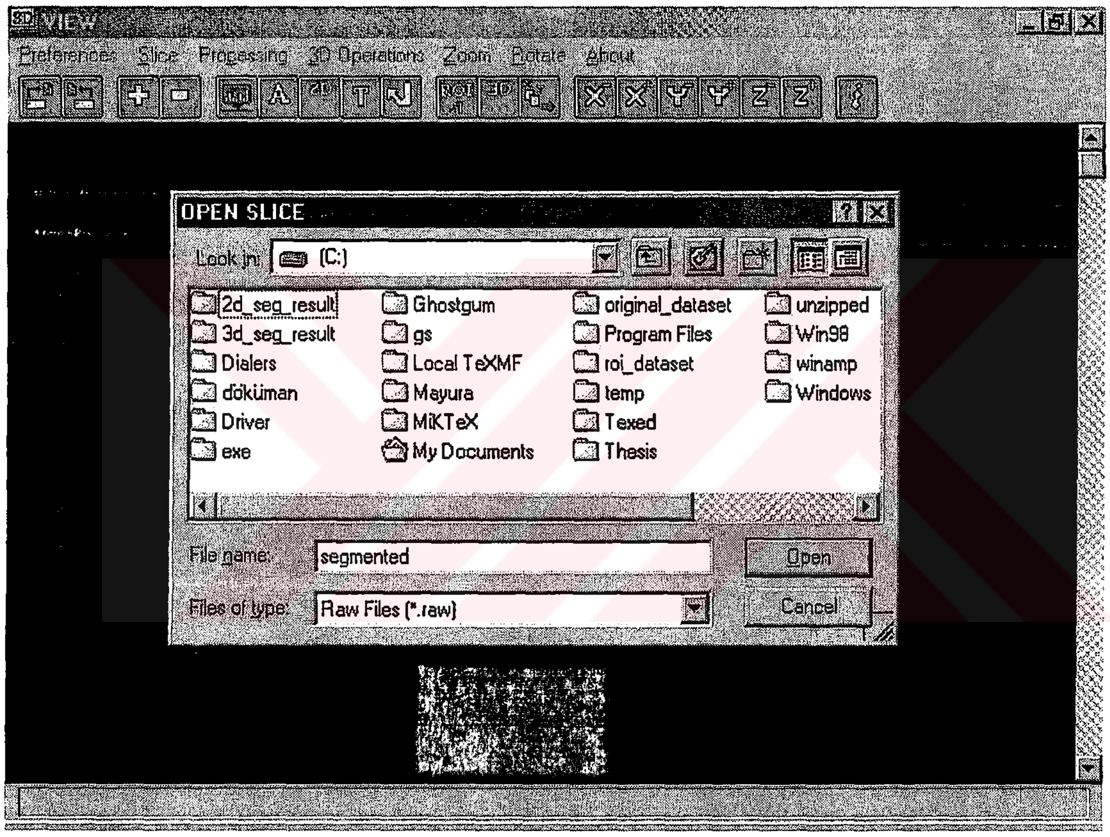
The Processing pop up menu in which the image processing functions take place is shown in Figure 6.7. The functions are undo operation, histogram equalization, low pass filtering, laplacian edge enhancement, thresholding, histogram show, 2-D segmentation, neigbourhood averaging, image negation and brightness adjustment. The way of implementing some of these functions and their implementation results will be given in the following parts.

Histogram equalization algorithm mentioned in Chapter 4, was applied to the original image shown in Figure 6.8. The result is shown in Figure 6.9. As it is seen from the image, the dark sides become clearer and it looks like the resulted image is giving more information than the original one. This is tricky since the algorithm actually leads to the loss of information while mapping the gray values into the new



Displays a slice from a dataset

Figure 6.4 Slice pop up menu and its menuitems.



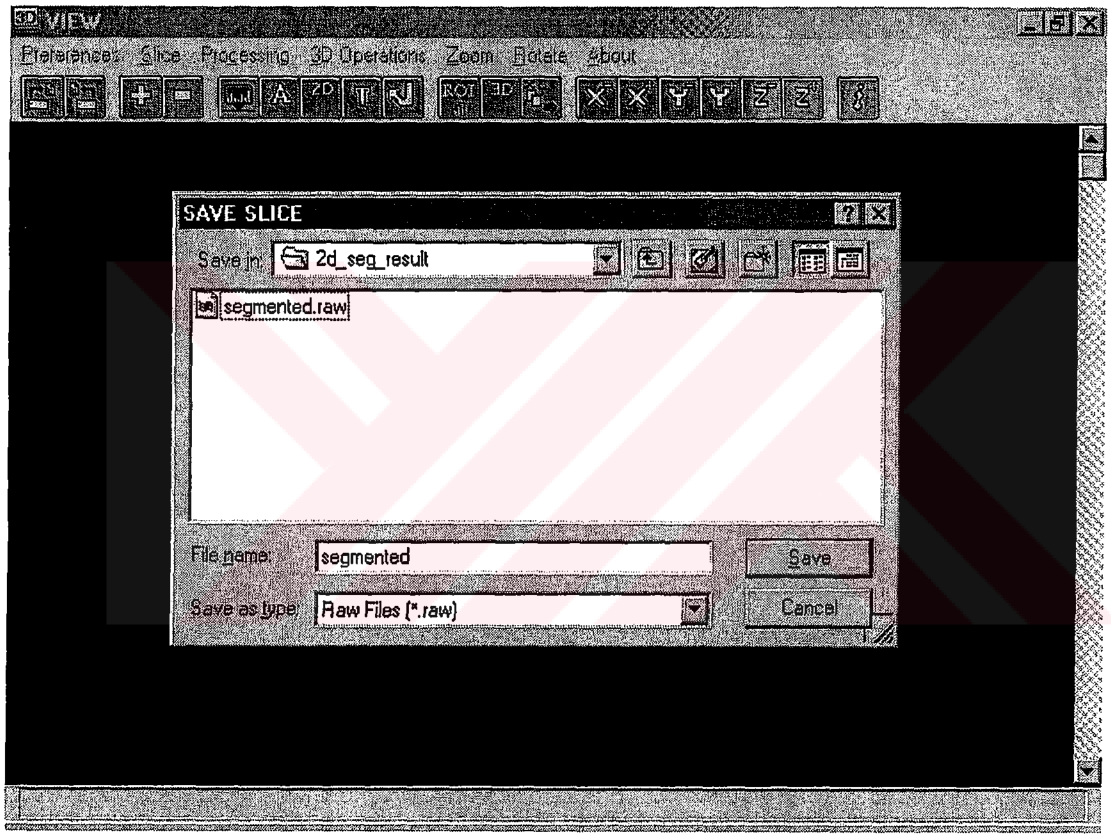
Look in: (C:)

2d_seg_result
3d_seg_result
Dialers
döküman
Driver
exe
Ghostgum
gs
Local TeXMF
Mayura
MiKTeX
My Documents
original_dataset
Program Files
roi_dataset
temp
Texed
Thesis
unzipped
Win98
winamp
Windows

File name: segmented

Files of type: Raw Files (*.raw)

Figure 6.5 Open Slice Common Dialog Box.



Save in: 2d_seg_result

File name: segmented

Save as type: Raw Files (*.raw)

Figure 6.6 Save Slice Common Dialog Box.

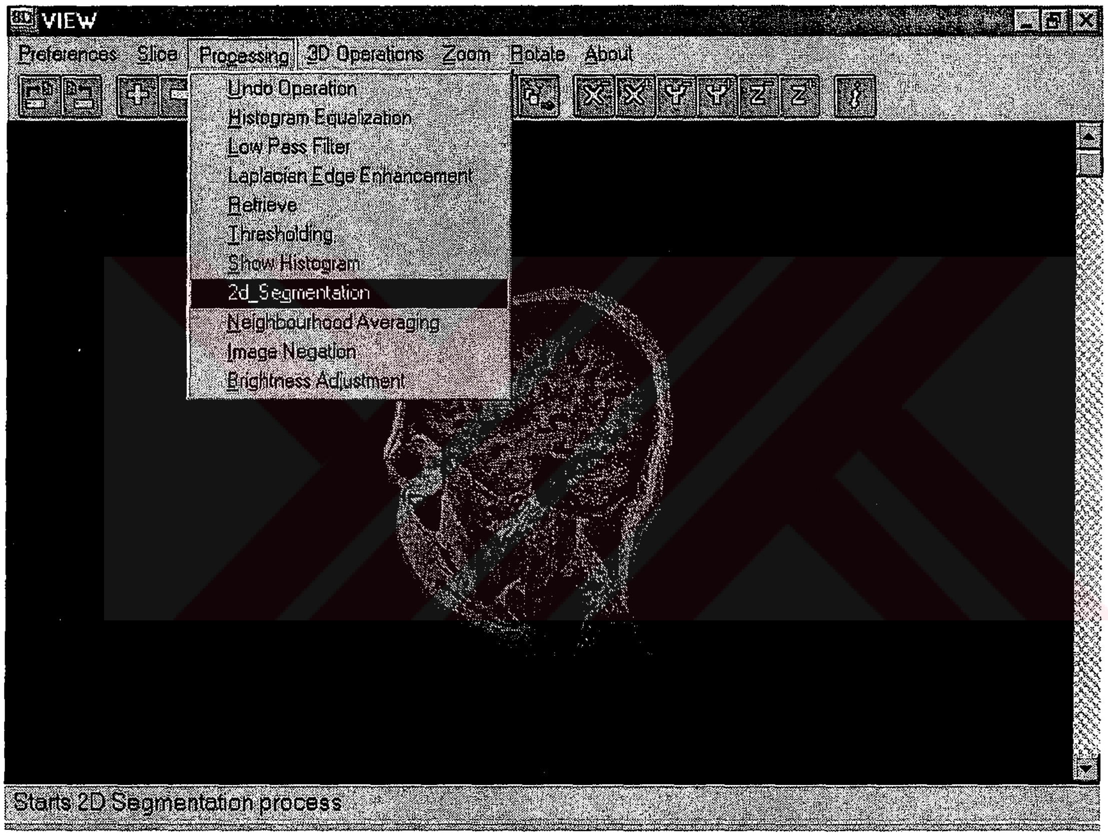


Figure 6.7 Processing pop up menu and its menuitems.

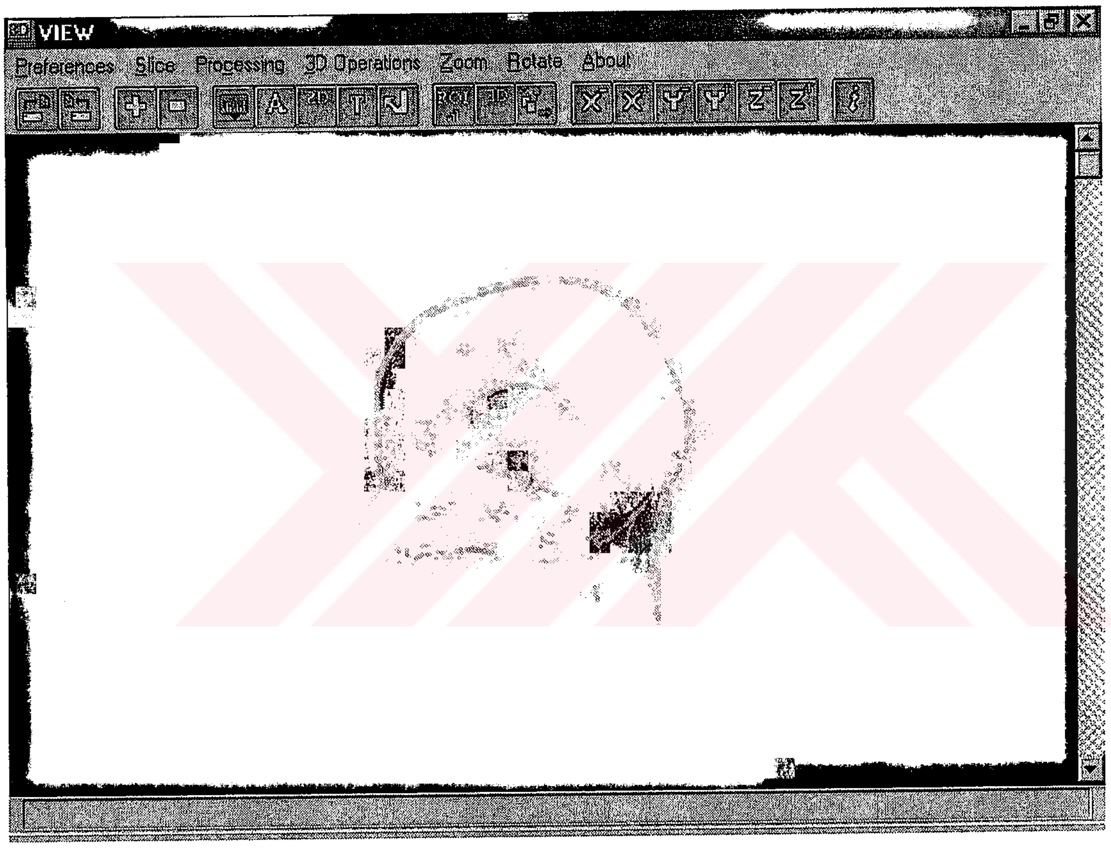


Figure 6.8 Original slice for the evaluation of processing results.

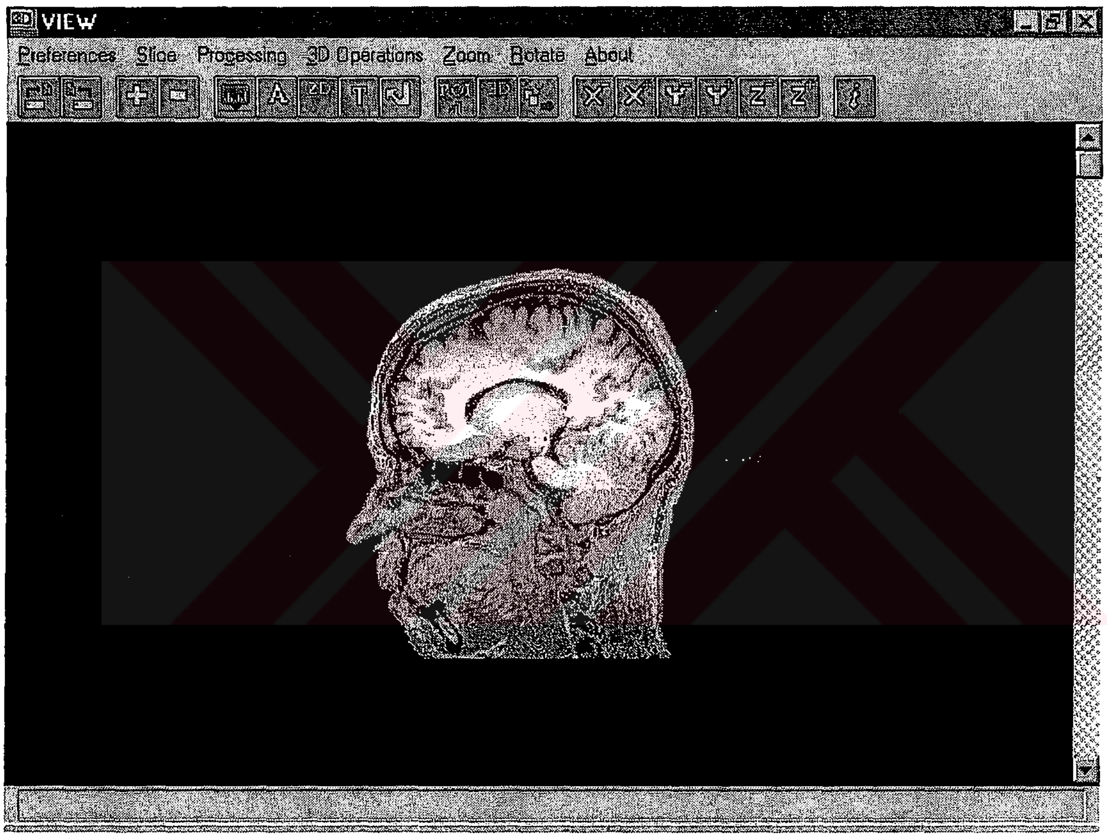
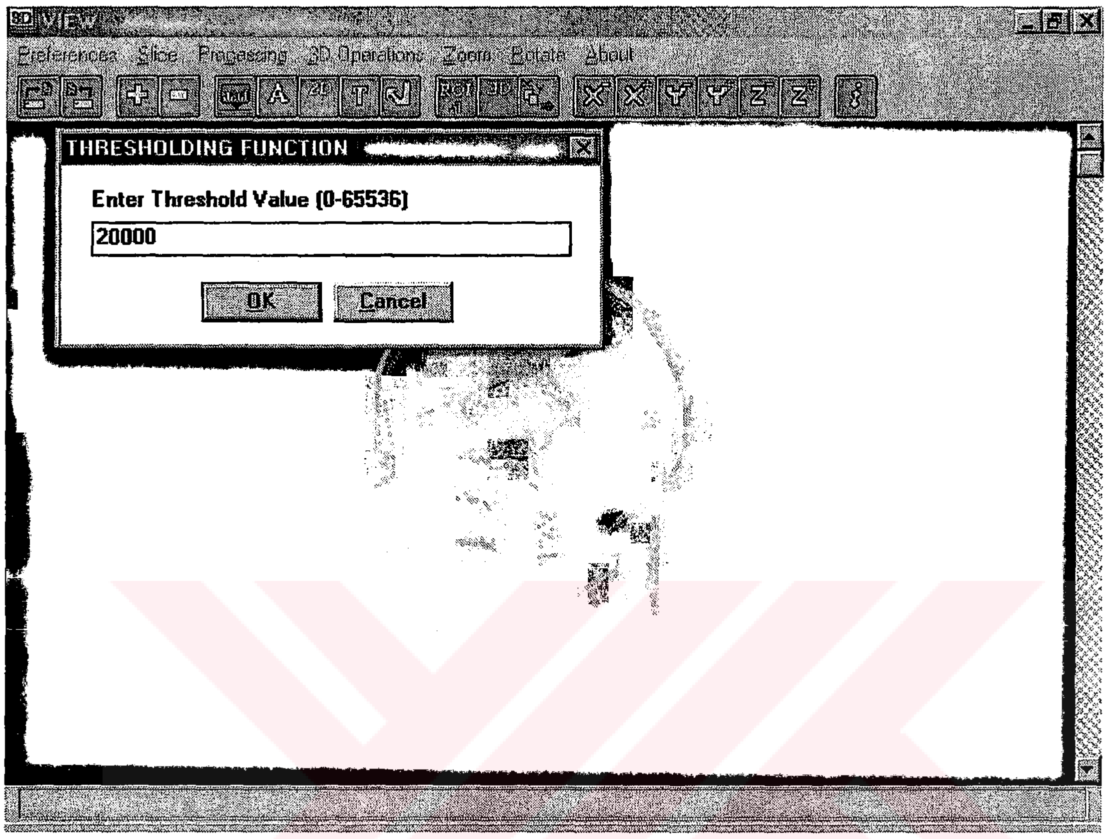


Figure 6.9 Histogram equalization process result.

ones. Actually, the algorithm spreads and moves the pixel grey values into more visible part of the histogram graphic.



Enter Threshold Value (0-65536)

20000

Figure 6.10 Thresholding input dialog Box.

As it is mentioned in Chapter 4, in detail, the thresholding function makes it possible to binarize the image. One threshold value is represented through the input dailog box shown in Figure 6.10. Since the original image data includes the grey levels from 0 to 65536 the threshold value must be in this interval.

The resulting image, passing 20000 grey value as a threshold parameter to the original image is shown in Figure 6.11. In the binarization process, below the threshold value was represented as 0 and the others are 65536.

When the image to be segmented is opened, if we select the 2-D segmentation menu-item from the process pop up menu, the 2-D segmentation process starts with asking the total seed count for three regions, for example white matter, gray matter and skull. This is shown in Figure 6.12. If the total seed count is 36 for example, 12

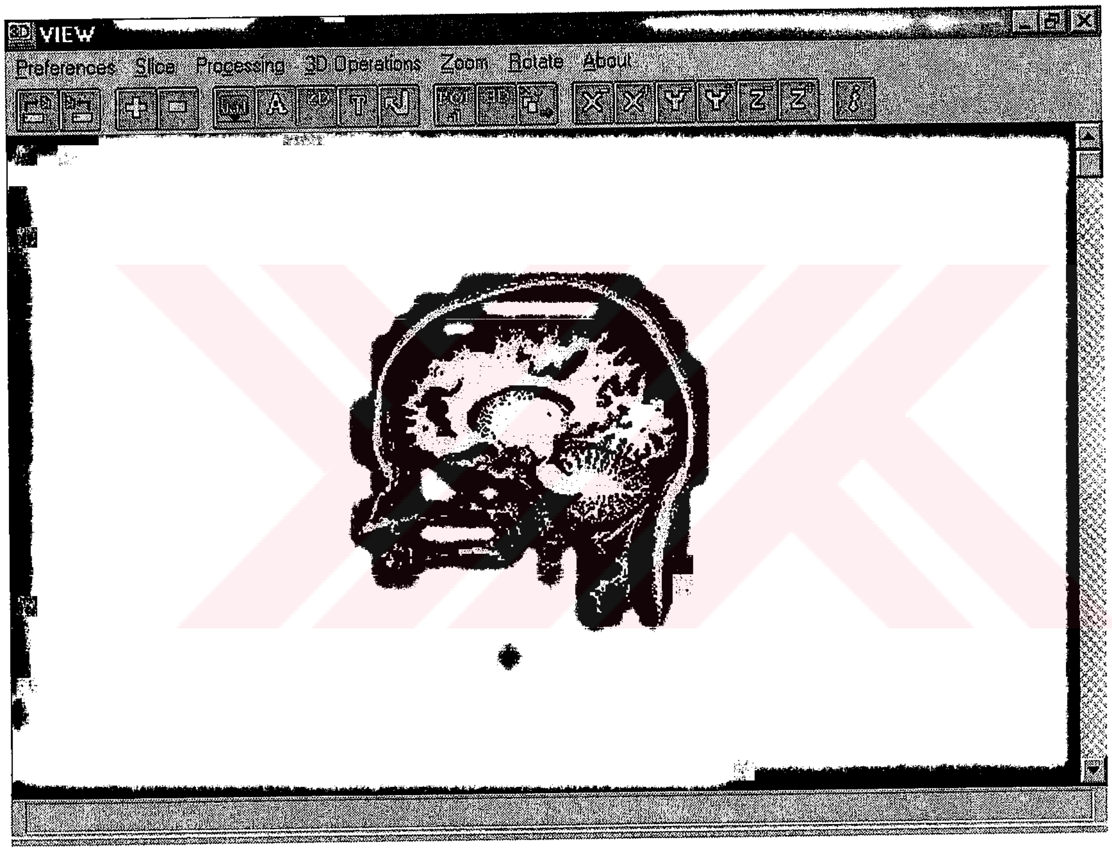
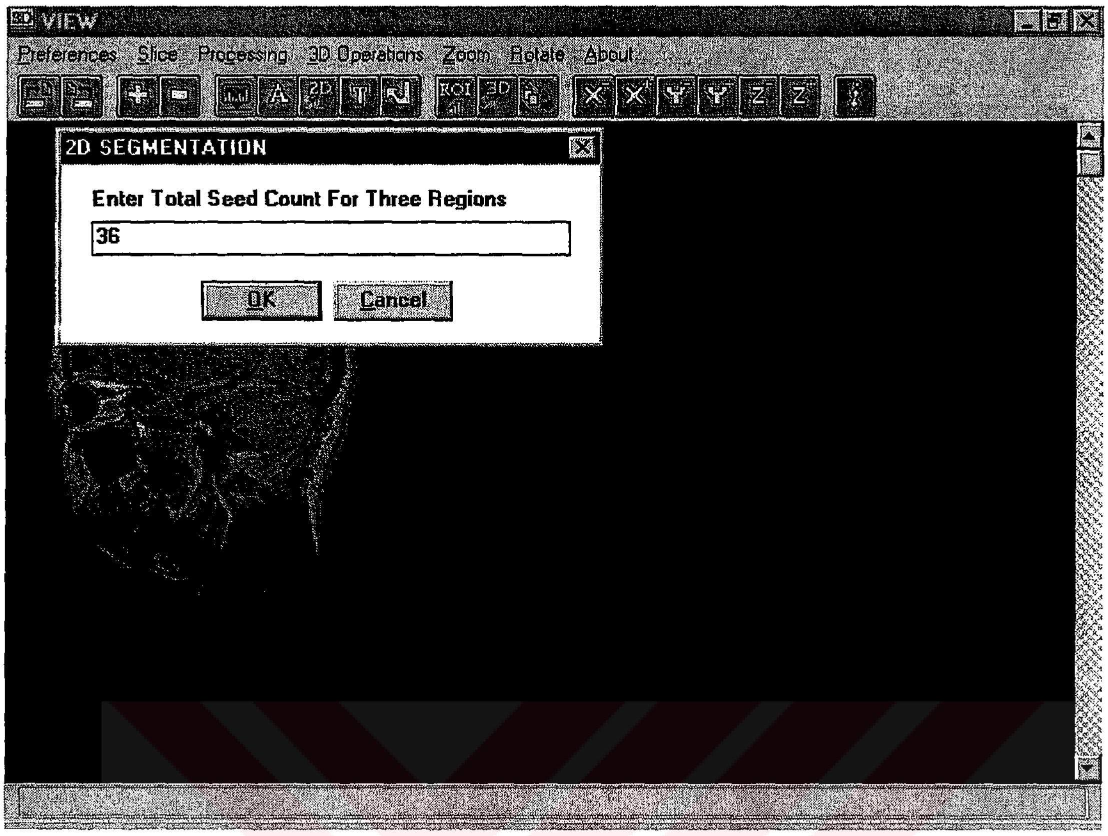


Figure 6.11 Thresholding process result.



Enter Total Seed Count For Three Regions

36

Figure 6.12 2-D segmentation seed count parameter representation.

seeds are represented for each region.

Seeds are represented interactively by clicking the left mouse while strolling through the image. A seeded image sample is shown in Figure 6.13.

The function applies the SRGA for 2-D segmentation. The result is shown in Figure 6.14 and the process time is approximately one and a half minute.

As it is mentioned in Chapter 4, 3-D segmentation was assumed as consecutive repeat of 2-D segmentation process. When the menu-item is selected, the toolbox asks for the total seed count for three region. These steps are shown in Figure 6.15 and in Figure 6.16 respectively.

Then the slice which is in the middle of the 3-D dataset, is displayed on the screen and the interactive seed representation starts. The toolbox realizes 2-D segmentation

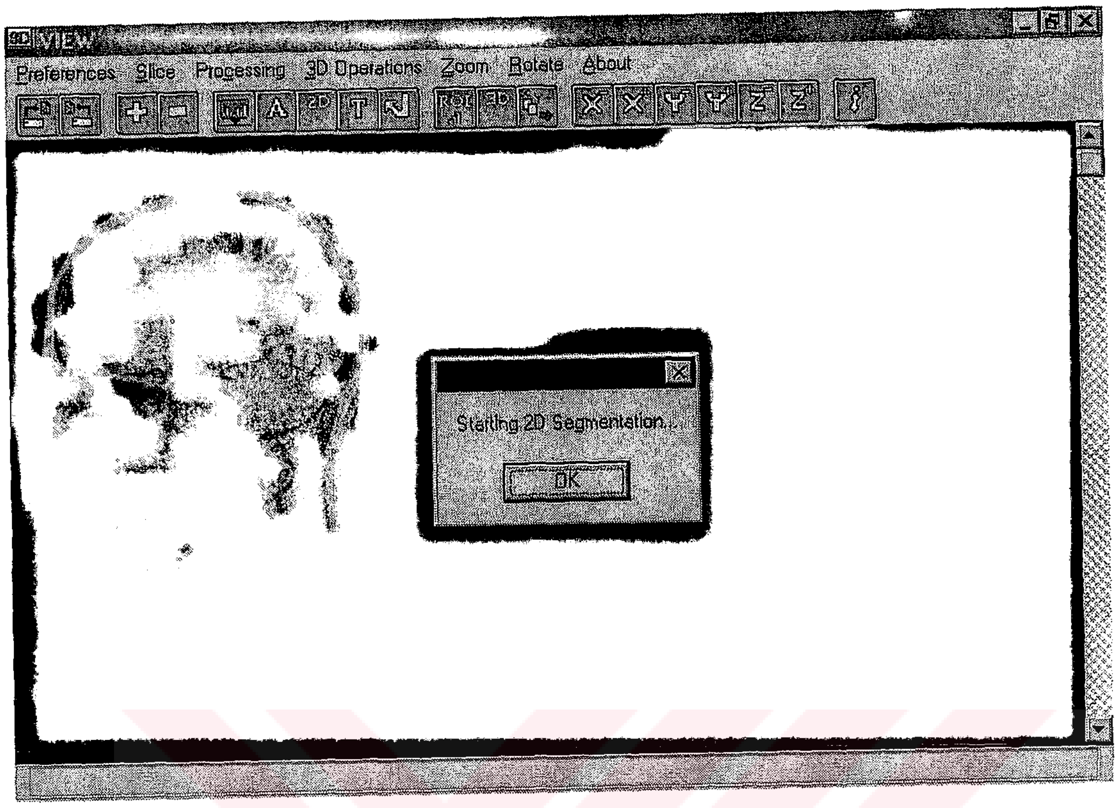

3D VIEW
Preferences Slice Processing 3D Operations Zoom Rotate About
Starting 2D Segmentation...
OK

Figure 6.13 2-D segmentation seed representation.

on this slices and determines the intensity intervals for each region by taking the seeds represented by the user into account. Then by taking these average intensity values, auto seeding function is called. This function is responsible for scanning the whole dataset and representing new seeds in every slice.

After the auto seeding function, the toolbox asks for the slice interval. The slices in this interval is subject to 2-D segmentation process with pre-defined seeds represented in auto seeding function. The result is kept in the 3-D segmentation destination directory given in path configuration.

Now the toolbox is ready to show the segmented images in 3-D manner. The region view utility is used for this purpose. This utility asks for the segmented region number and the segmented slice interval. The result is shown in Figure 6.17.

In order to see the whole dataset in 3-D manner, a function called slice by slice

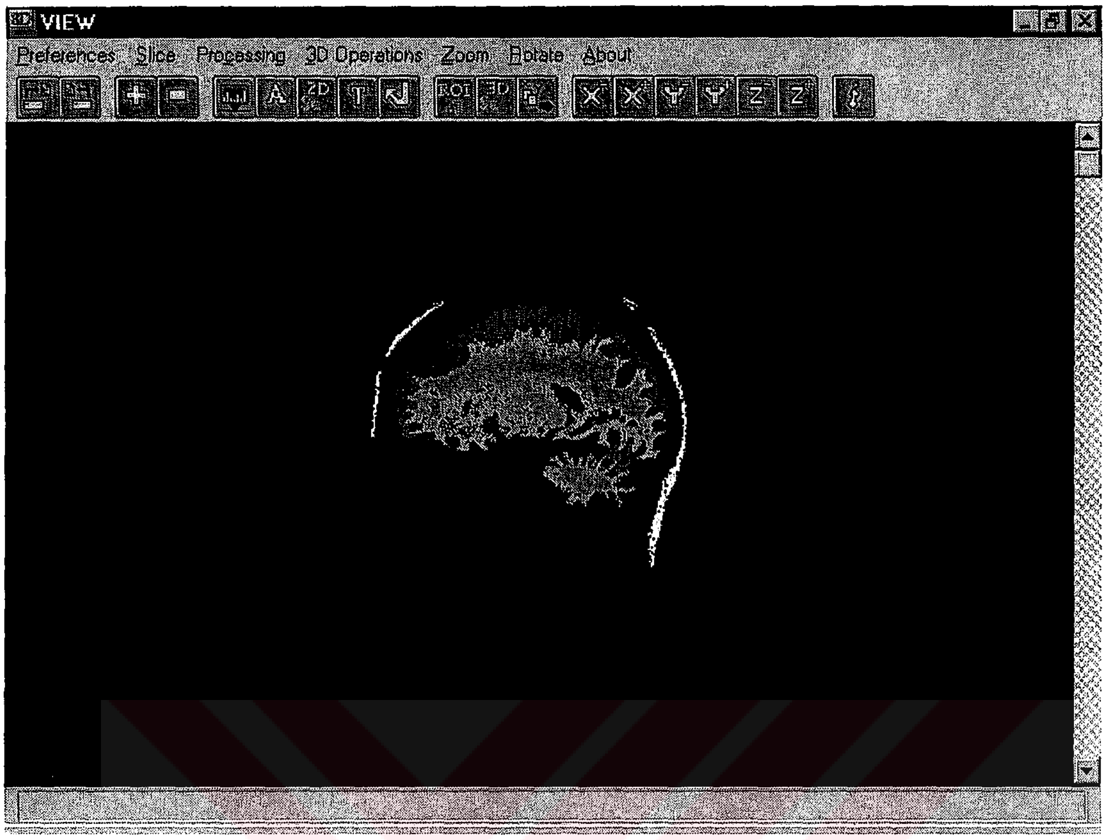

VIEW
Preferences Slice Processing 3D Operations Zoom Rotate About

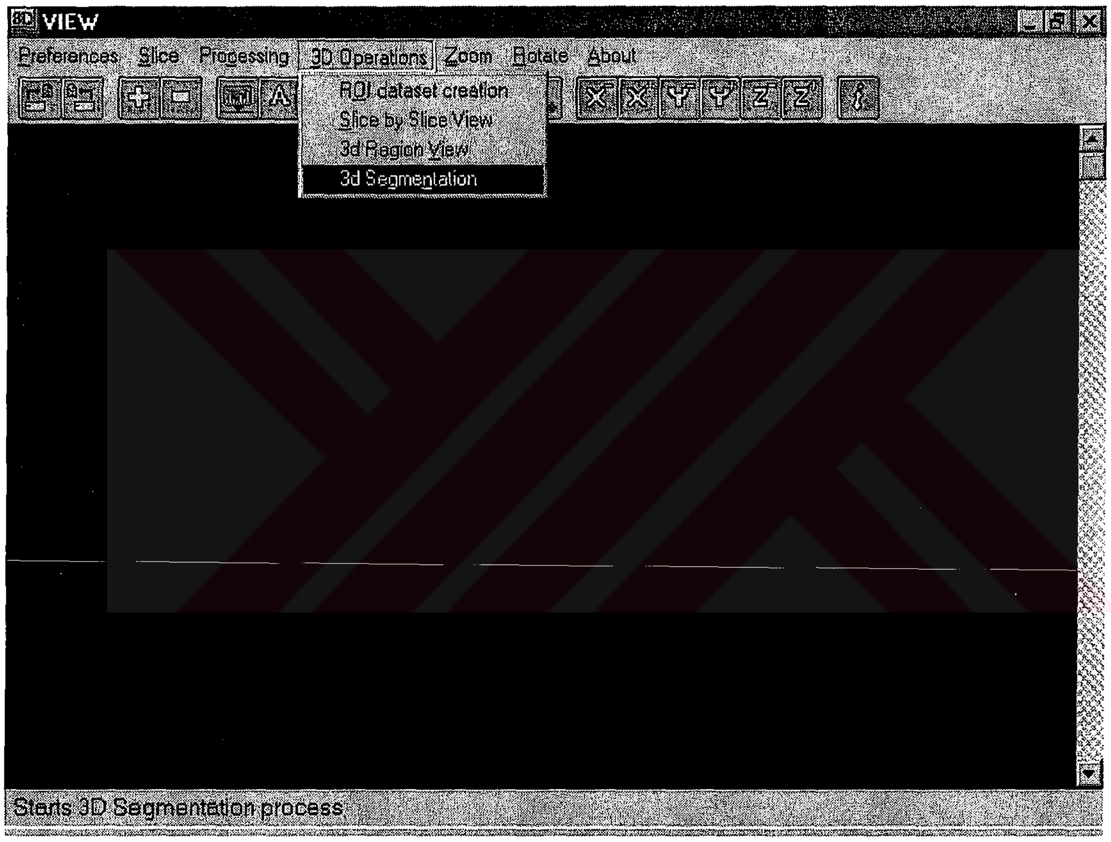
Figure 6.14 2-D segmentation process result.

view was written. This function call and the result for slice 40 and slice 60 interval (20 slices) are shown in Figure 6.18.

If we choose the interval between 0 and 127, we obtain the full dataset, viewing in three dimensions. It is possible to have different camera views by using the utilities such as zooming in and zooming out or rotation about x, y, or z axis. These utulities are shown in Figure 6.19 and in Figure 6.20 respectively. The function responds these commands by handling the windows event messages.

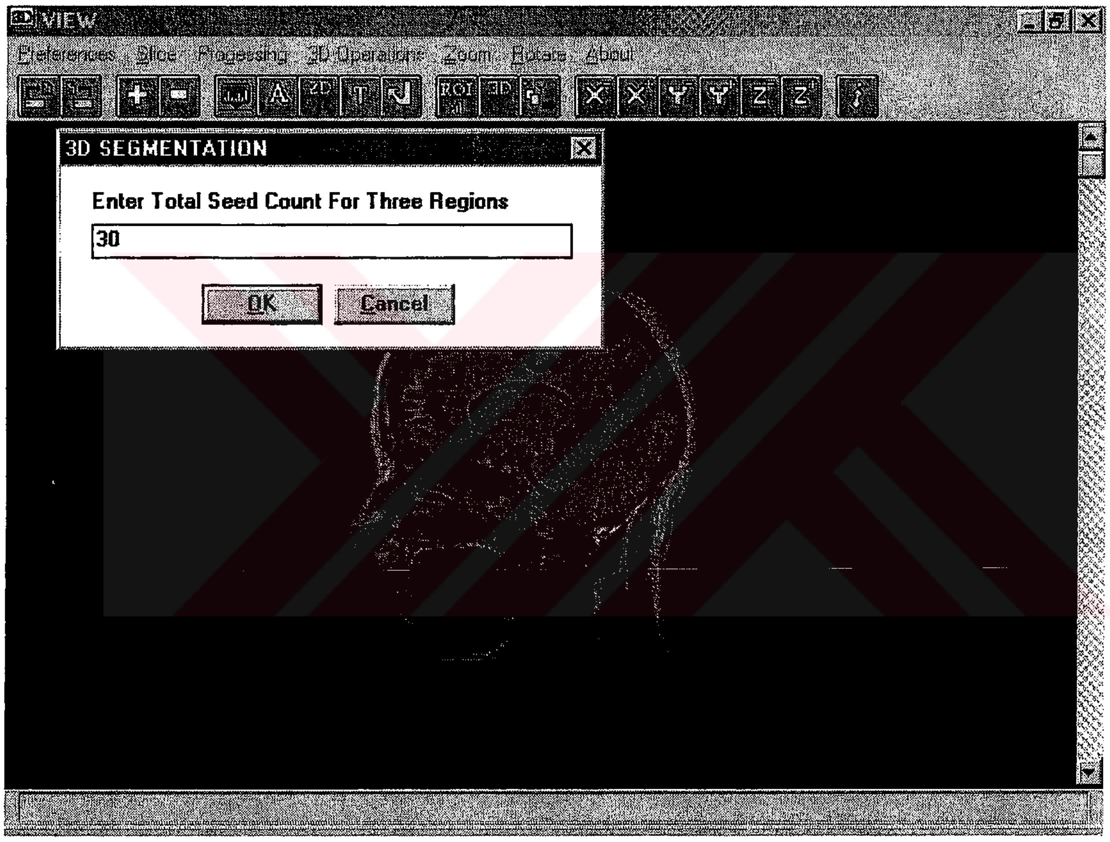
When the slice count is increased, the process time also increases. It is clear that instead of using classical matrix transformations for three dimensional view and rotation, using OpenGL makes the process time efficent. This is done by the quick interface between the device context (graphics card) and the software, namely OpenGL.

The slice interval 20-100 3-D rendering result is shown in Figure 6.21.



Starts 3D Segmentation process

Figure 6.15 3-D segmentation process startup.


VIEW
Preferences Slice Processing 3D Operations Zoom Rotate About
OK
Cancel

3D SEGMENTATION

Enter Total Seed Count For Three Regions

30

Figure 6.16 3-D segmentation seed representation.

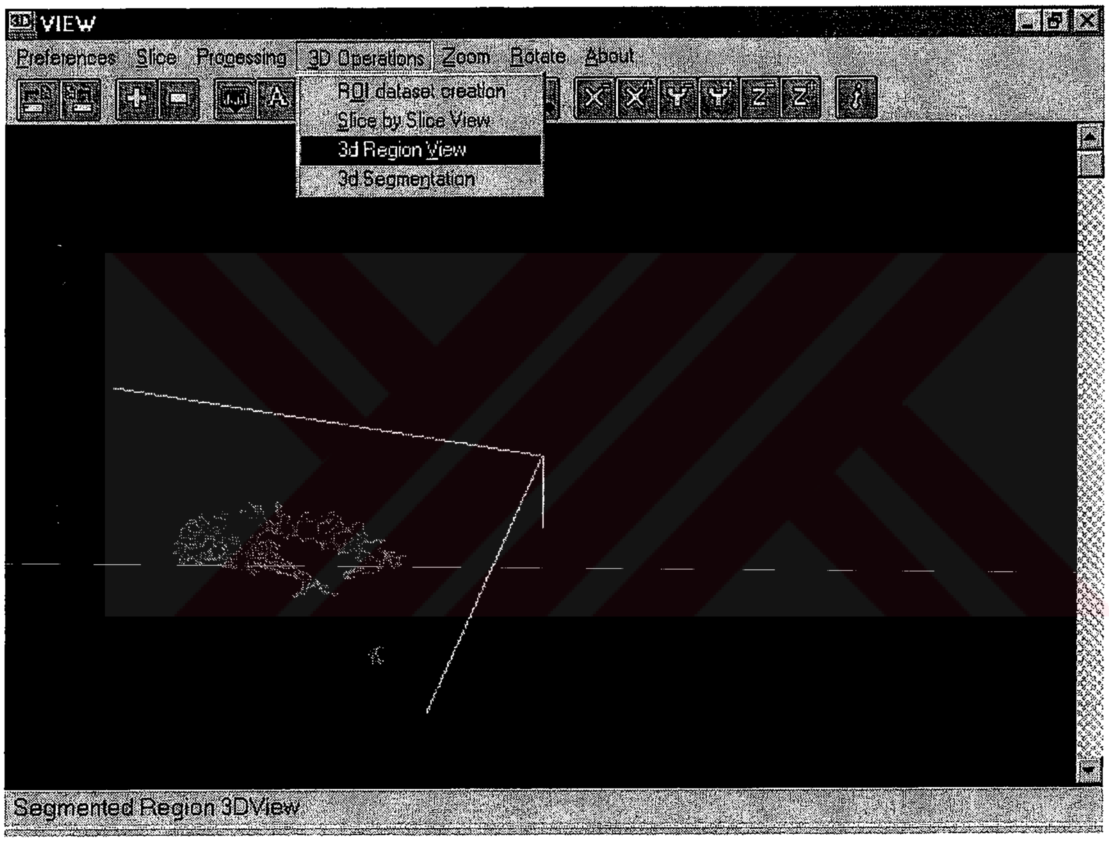
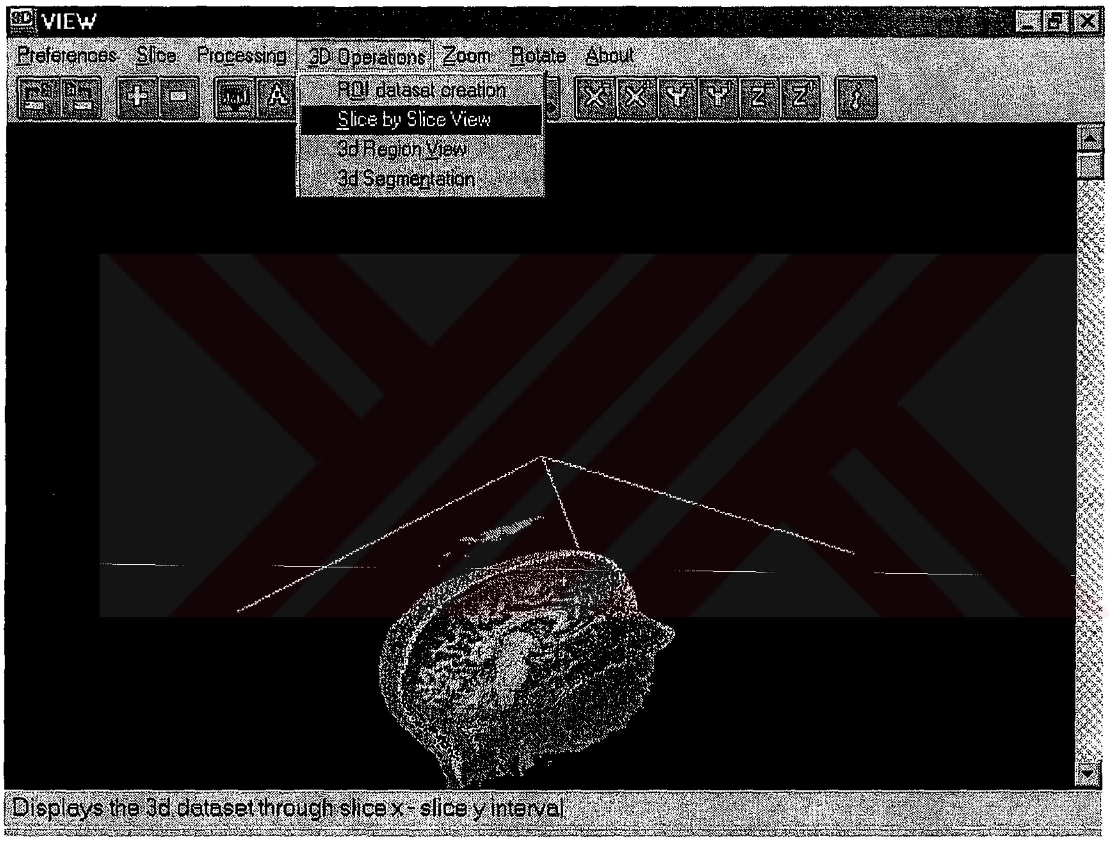


Figure 6.17 3-D segmented region view, white matter.



Displays the 3d dataset through slice x - slice y interval

Figure 6.18 Slice by Slice menu-item and result.

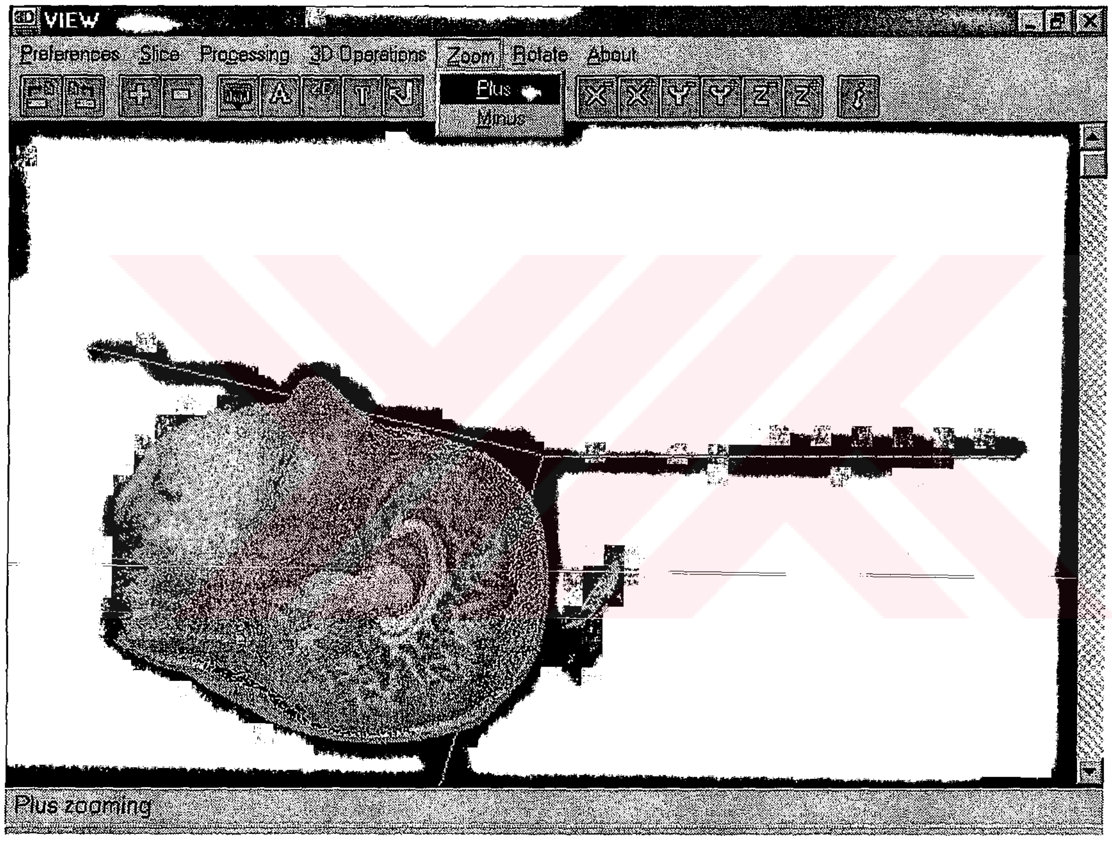
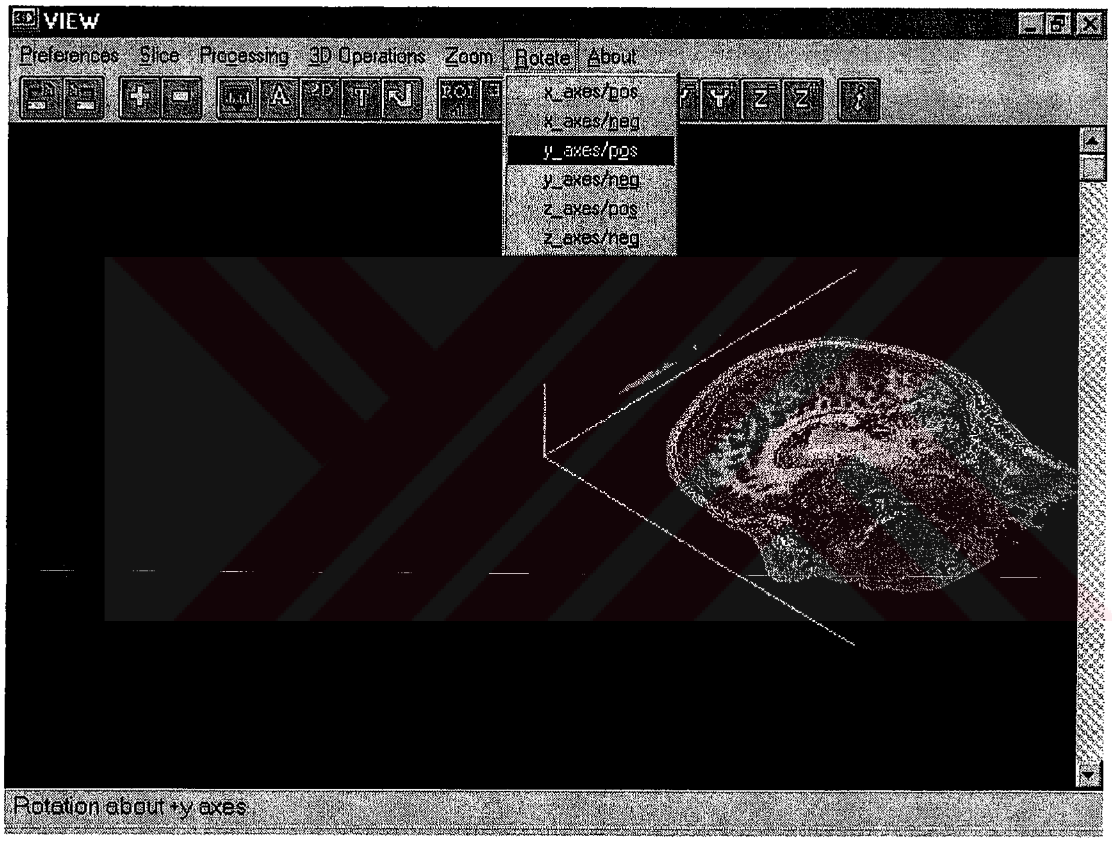


Figure 6.19 3-D zooming menu and result.



Rotation about +y axes

Figure 6.20 3-D Rotation menu and result.

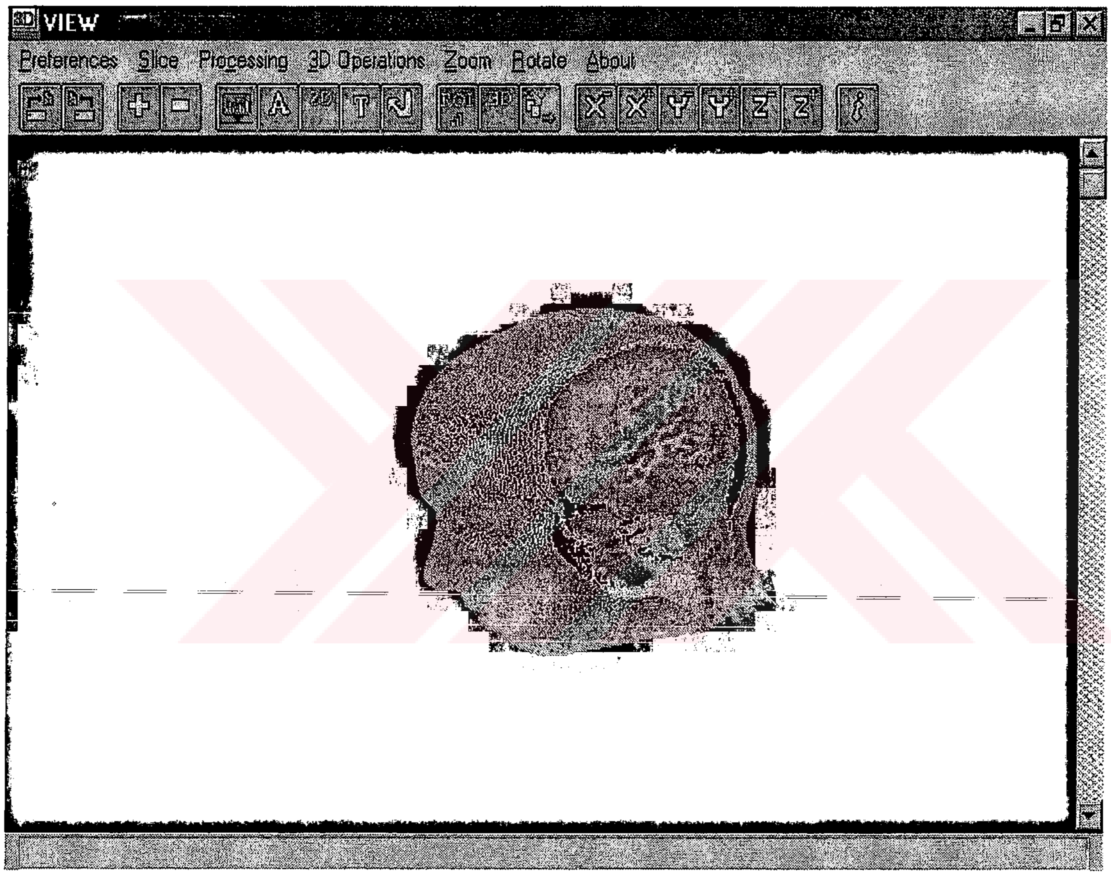


Figure 6.21 3D view of 20-100 slice interval.

7. DISCUSSIONS, CONCLUSIONS and FUTURE WORK

For the implementation, menu driven graphical user interface resources such as dialog boxes, bitmaps, pop-up menu and menuitems have been created by the resource generator and compiler, BRW. Whole code has been written in Borland C/C++ programming language using ObjectWindows 2.0. For the compilation, Borland 32-bit C/C++ compiler version 4.5 has been used.

The toolbox functions are:

- 2-D Display
- 3-D Display
- 2-D Image Processing
- 3-D Image Processing
- 2-D Segmentation
- 3-D Segmentation
- 3-D Segmented Region View

The toolbox makes it possible to display and process 3-D biomedical images by taking advantage of OpenGL. The OpenGL is created for computer graphics applications and it facilitates the matrix calculations and matrix operations. This leads to a remarkable enhancement in the software performance. The user may view the related slice in 2-D or 3-D manner or may display some set of slices in 3-D. The toolbox can render some portion or the whole biomedical volume and can make it possible to apply some 3-D operations such as 3-D rotations in x, y and z coordinates, 3-D zooming. Since the OpenGL library handles all the matrix conversions that are necessary for translation, scaling and rotation operations, there is no need to deal with some matrix multiplication and calculations. If the amount of matrix calculations are taken into account, it is obviously confirmed that OpenGL increases the system performance and the processing speed.

For the processing part, basic image processing algorithms such as histogram equalization, image negation, brightness adjustment, thresholding, filtering, edge enhancement, 2-D and 3-D segmentation have been implemented. All these functions are in modular structure and new Windows resources such as pop-up menu, menuitem, button gadget(menu-item shortcut) or an algorithm function can be inserted into the toolbox by making some modifications on the software. The user may insert a new algorithm to the toolbox by just following the methods explained in Chapter 5 in detail. He/she may observe the algorithm results and make algorithm performance analysis by using the toolbox facilities. By following the methods for creating new Windows resources such as dialog box, button gadget, pop-up menu, menuitem, the user may add new functionalities to the toolbox.

The user may open a specific slice from a dataset and after the modification, that slice can be saved in pre-defined directory. All the working directory paths for the input and the output files can be set in the toolbox.

The required minimum random access memory capacity is 16M. The graphics card's memory should not be less from 1M and a fast processor is recommended.

The source code only supports the biomedical images with the .RAW extension. This extension is representing the binary coded imaging method. For future work, the toolbox may handle different types of image formats such as JPEG, BMP, GIF, TIFF.

The toolbox can be upgraded so as to work in Windows 2000/NT or ME edition operating systems. Since there are some restrictions coming from the OpenGL part, the programme can not work in Windows 95/98 and Windows 2000 operating systems at the same time. The compatibility for the Windows 2000 based operating systems can be provided by building a flexible software which detects the operating system automatically. Alternatively, a new executable application file can be built for the Windows 2000 based operating systems.

For future work, the source code can be made flexible regarding the size of biomedical image dataset. The code can be modified to support different size of datasets. This can be done by utilizing a new dialog box asking the width and height of the image in Open Slice menuitem. The size of the image can be introduced to the toolbox via that dialog box. Alternatively, by taking the extension of the image into account and by fetching the appropriate bytes from the image data, the image size can be determined automatically by the software.

APPENDIX A. DEVELOPED SOFTWARE LISTING

A Compact Disk (CD) comprising the source code listing, used slice dataset, header files, OpenGL related library files, executable application file, required DLL files and related BRW source files has been included. CD also contains a file called "READ.ME" in ASCII text file format, containing the sections mentioned below:

- Files in the compact disk
- The programme hardware requirements
- The programme software requirements

Files in the compact disk:

1) gl.exe : Executable 3DVIEW application file.
2) int1.cpp : C++ source code.
3) Opengl.dll : Opengl dynamic link library file.
4) Opengl32.dll : Opengl dynamic link library file for 32-bit applications.
5) Dciman.dll : Necessary dynamic link library file.
6) Dciman32.dll : Necessary dynamic link library file for 32-bit applications.
7) trial.rc : Resource file for the BRW programme.
8) trial.res : Resource file for the BRW programme.

9 trial.rws : Resource file for the BRW programme.

10) Tglwind.h : Header include file for int1.cpp file.
11) Cmdlg.h : Header include file for int1.cpp file.

12) Cmdlgr.h : Header include file for int1.cpp file.

13) Ids.h : Header include file for resource identifiers.

14) Gl.h : Header include file for OpenGL.

15) Glu.h : Header include file for OpenGL.

16) Glaux.h : Header include file for OpenGL.

17) Glut.h : Header include file for OpenGL.

18) openGl.lib : Library file for OpenGL.

19) opengl32.lib : Library file for OpenGL for 32-bit applications.

20) Glu.lib : Library file for OpenGL.

21) Dataset: Real human head dataset comprsing 128 slices.

22) README.txt : Readme text file.

Hardware Requirements:

The required minimum random access memory capacity is 16M. Display card screen area resolution should be chosen as 640x480.

Software Requirements:

Toolbox software has been tested on Windows 95 and Windows 98 operating system platforms.

REFERENCES

1. Rogers, D., and J. Adams, *Mathematical Elements for Computer Graphics*, New York: McGraw-Hill, 1976.

2. Foley, J., and A. V. Dam, *Fundamentals of Interactive Computer Graphics*, Mass.: Addison-Wesley, 1982.

3. Blinn, J., and M.E.Newell, "Clipping using homogeneous coordinates," *Computer Graphics*, Vol. 12, pp. 245–251, August 1978.

4. Newman, W., and R. Sproull, *Principles of Interactive Computer Graphics*, New York: McGraw-Hill, 1979.

5. Woo, M., and J. Neider, *OpenGL Programming Guide*, Addison-Wesley, 1997.

6. Adams, R., and L. Bischof, "Seeded region growing," *IEEE Trans. Pattern Anal. Machine Intelligence*, Vol. 16, no. 6, pp. 641–647, 1994.

7. Yulong, Y., and X. Bao, "3-d imaging and stereotactic radiosurgery," *IEEE Engineering in Medicine and Biology*, Vol. 739, no. 5175, 1997.

8. Thomas, S., and U. Tiede, "Visualization blackboard," *IEEE Computer Graphics and Applications*, Vol. 16, no. 17, p. 272, 1996.

9. Verdina, J., *Projective Geometry and Point Transformations*, Boston: Allyn and Bacon, 1971.

10. Robb, R. A., and C. Barillot, "Interactive display and analysis of 3-d medical images," *IEEE Trans. Med. Imag.*, Vol. 8, no. 217, 1989.

11. Jackway, P., and A. Mehnert, "An improved seeded region growing algorithm," 18 1065-1071, Elseiver Pattern Recognition Letters, 1997.

12. Anton, H., *Linear Algebra*, New York: Wiley, 1981.