AVW Programmer's Guide

Biomedical Imaging Resource
Mayo Foundation
Rochester, Minnesota

Staff of the Biomedical Imaging Resource, Mayo Foundation
Richard A. Robb, Ph.D., Director
Kurt Augustine
Darlene Bernard
Bruce Cameron
Jon Camp
Dennis Hanson
Ron Karwoski
Mark Korinek
Al Larson
Russ Moritz
Margret Ryan
Mahlon Stacy
Ellis Workman

Other Contributor
Dan Blezek
Armando Manduca

Advisor / Publisher
Richard A. Robb, Ph.D.

CHAPTER 1 ***Introduction*** ***1***

What is AVW? **1**

Overview of AVW **2**

CHAPTER 2 ***AVW Concepts*** ***5***

AVW_Image **5**

Image Memory ***6***

Data Types ***6***

Image Dimensions ***7***

Precomputed Values ***7***

Color Lookup Table ***7***

Other Information ***7***

Image Offsets ***8***

AVW_Volume **9**

Create and Destroy **9**

Memory Usage **11**

Color Images **12**

Points and Point Lists **13**

Volume Coordinate System **15**

Function Specification and Calling Conventions **15**

Error Handling **16**

Man Pages **16**

ANSI C Prototyping **16**

Compiling and Linking **16**

CHAPTER 3 ***Image I/O*** ***17***

Similarity to UNIX File I/O **17**

The Data Structures **18**

AVW_ImageFile ***18***

Extending to Other Formats ***19***

AVW Image I/O Functions **20**

Properties **21**
AVW Info Strings **23**
Image File Formats **24**
Special AVW Image File Formats **25**
AVW_VolumeFile Format ***25***
AVW_ImageFile Format ***28***
Reading Raw Data into an AVW_Image **29**
Extending I/O for Other Image File Formats **31**
Example Programs **33**
TCL-TK Implementation **34**

CHAPTER 4 *Transform Functions* *37*

Plane Extraction **37**
Orthogonal Planes ***37***
Oblique Planes ***38***
Curved Planes ***39***
Basic Manipulations **40**
Copying ***40***
Setting to a Value ***40***
Flipping ***41***
Shifting ***41***
Conversion **41**
Changing Data Types ***42***
Thresholding ***43***
Intensity Scaling ***43***
Dithering ***44***
Conversion to Gray Scale ***45***
Inverting ***45***
Applying Tables ***45***
Image to Volume ***45***
Padding, Subregioning, and Resizing **45**
Padding ***46***
Subregioning ***46***
Resizing ***46***
Image and Volume Mathematics **47**

Basic ***48***
Boolean ***48***
Logical ***48***
Functions ***48***
Constants ***49***
Images ***49***
Matrix Manipulations **50**
Identity Matrix ***51***
Matrix Manipulations ***51***
Applying a Matrix to an Image ***52***

CHAPTER 5 *Process Functions* ***53***

Histogram Operations **53**
Histogram Matching ***53***
Histogram Flattening ***55***
Histogram Preservation ***56***
Spatial Filter Operations **57**
Spatial Convolution ***57***
Simple Convolution Filters ***59***
Other Edge Filters ***61***
Non-Linear Filters ***62***

CHAPTER 6 *Fourier Transforms and Deconvolution* ***67***

Fourier Transforms **67**
Frequency Space Convolution **67**
Filter Creation **68**
Deconvolution **69**
Simple thresholding ***69***
Wiener Deconvolution ***70***
Constrained Iterative Deconvolution ***70***
Nearest Neighbor Deconvolution ***71***
Sample programs ***71***

CHAPTER 7 *Segmentation* ***73***

Thresholding **73**
Region Growing **74**
Mathematical Morphology **75**
Other Segmentation Routines **77**
Sample Code **77**

CHAPTER 8 *Analysis* ***79***

Line Profiles **79**
Masks **80**
Histograms **81**
Intensity Statistics **82**
Shape Statistics **86**
Sample Code **89**

CHAPTER 9 *Visualization* ***91***

Render Parameters **91**
- *Render Type* ***93***
- *Thresholds* ***96***
- *Translation and Rotation* ***96***
- *Scale* ***97***
- *Clip Volume* ***97***
- *Clip Plane* ***97***
- *Clip Shading* ***97***
- *Render Buffer Dimensions* ***97***
- *Render Values* ***98***
- *Lighting* ***98***
- *Render Mask* ***98***
- *Perspective Rendering* ***98***
- *Specular Reflection* ***99***

Post Processing **99**
Interface **99**

Interactive Surfaces **100**
Clean Up **101**

CHAPTER 10 *Object Maps* ***103***

AVW_ObjectMap Structure **103**
AVW_Object Structure **104**
Creation and Destruction of Object Maps **106**
Saving an Object Map to Disk **107**
Object Map File Format **107**
Note on Version Numbers ***110***
Loading a Previously Defined Object Map **111**
Creating New Objects **112**
Deleting Objects **112**
Extracting an Object from an Object Map **112**
Using a AVW_Volume to Define an Object **112**
Using Ray-Casting to Define an Object **113**
Using Region Growing to Define an Object **113**
Manipulating the Mapping Volume with Other AVW Routines **114**

CHAPTER 11 *Multi-Spectral Classification* ***115***

Classification Algorithms **115**
Input Requirements **116**
Classify Functions **117**
Training Images and Volumes **119**
Control Parameters **120**
Scattergrams **121**
Sample Programs **123**
References **123**
124

CHAPTER 12 *Surface Matching* ***125***

AVW_MatchParameters Structure **125**
AVW_MatchResult Structure **126**
Calling AVW_MatchSurfaces() **127**
Sample Code **131**
References **131**

CHAPTER 13 *Modeling* ***133***

The Algorithms **133**
- *Kohonen Based Tiling* ***133***
- *Growing Networks* ***134***
- *Post-Processing* ***135***

Tiling Parameters **136**
- *Tile Type* ***138***
- *Tile Mask* ***138***
- *Tile CurveOpRadius* ***139***
- *Tile CloseSrfcFlag* ***139***
- *Tile KohonenMajorAxis* ***139***
- *Tile KohonenShapeOrient* ***140***
- *Tile KohonenShapeOffset* ***140***
- *Tile KohonenFlag* ***140***
- *Tile PolygonBudget* ***141***
- *Tile KohonenRepetitions* ***141***
- *Tile KohonenNeighborhood* ***141***
- *Tile KohonenNeighborRadius* ***142***
- *Tile KohonenAlpha* ***142***
- *Tile AddNodeFreq* ***142***
- *Tile MaximumAge* ***142***
- *Tile Eb and En* ***143***
- *Tile GrowingAlpha and GrowingD* ***143***

Tiled Surfaces **143**
- *AVW_TileSurface Structure* ***143***

Creation and Destruction of Tiled Surfaces **144**
Saving a Tiled Surface to Disk **145**
Loading Polygonal Data as a Tiled Surface **146**

Interface **147**
Slice-based Surface Representations **148**
Contour Extraction **149**
Contour Parameters **149**
Contours MaskValue ***150***
Contours Format ***150***
Contours Orientation ***150***
Contours SubvolumeFlag ***150***
Contours InterpolateFlag ***150***
Contours AngleResolution ***151***
References **151**
Appendix **151**

CHAPTER 14 *AVW & Tcl/Tk* ***159***

What is Tcl? **159**
What is Tk? **160**
How do AVW and Tcl go together? **161**
What about AVW and TK? **161**
Imaging Program Pieces: Loading an Image **161**
Displaying an Image **162**
24 bit displays. ***163***
Adding AVW Image Processing Functions **163**
Interacting with Boxes and Buttons **164**
Adding a C function to your Program **166**
Saving an Image **167**
Sample Code **168**

CHAPTER 15 *AVW++* ***169***

Classes **169**
Example Code **170**

List of Figures ***173***

Index ***179***

CHAPTER 1

Introduction

1.1 What is AVW?

AVW (**A V**isualization **W**orkshop) is a comprehensive library of imaging functions which permits full exploration and analysis of multidimensional and multimodal biomedical image data sets. The software facilitates the development and implementation of advanced imaging algorithms and techniques and easy integration of these into specific applications solutions by imaging software developers. The extensive functionality and interactive speed of this software are unequivocally the key features which distinguish it from other image processing and visualization packages.

The **AVW** imaging functions are packaged into a library that provides developers with the power of its predecessor, the **ANALYZE**™ system, at a callable C language function level. The functions have been entirely rewritten from scratch, providing increased performance and flexibility. It is extensible, allowing developers to write their own specialized code while still making full use of the imaging capabilities of the package. The functions are completely user-interface independent, allowing for implementation at the applications level in different interface and operating environments. Each library function has a well defined calling sequence utilizing standardized **AVW** parameters and data structures.

1.2 Overview of AVW

Table 1-1 groups the **AVW** functions according to their functionality and lists the chapter in which the group is discussed.

TABLE 1-1 Function Groupings

Function Group	Description	Chapter
Resource	Basic routines to create and destroy many of the **AVW** structures. Also included are list management, verification, and byte swapping routines.	Chapter 2
Image I/O	Functions which provide a user extendable interface which allows the reading and writing of images in many common formats.	Chapter 3
Transform	Functions include 2D and 3D spatial transformations, densitometric transformations and mathematical transformations. The functions necessary for oblique and curved sectioning, wavelet enhancement transformations, and image registration/fusion are also included.	Chapter 4 Chapter 12
Process	Functions provide a wide variety of histogram operations, spatial/convolution filtering, 3D FFTs, 3D Fourier domain filtering and deconvolution, including fast nonlinear iterative methods.	Chapter 5 Chapter 6
Segment	Functions include thresholding, 2D and 3D region growing, automated boundary detection, creation of object maps, multi-resolution decomposition, morphological processing and multi-spectral classification.	Chapter 7 Chapter 10 Chapter 11
Analysis	Functions include extraction of line profiles, sampling of ROIs which includes the computation of size, density, shape and texture parameters (e.g., area, mean, circularity, fractal signature).	Chapter 8

TABLE 1-1 Function Groupings

Function Group	Description	Chapter
Visualize	Volume rendering routine includes depth-only shading, depth gradient shading, voxel gradient shading, integrated surface projection, maximum intensity projection, summation projection and perspective rendering. Functions to support multiple object creation and rendering, color transparency (24-bit), image extraction tools, and image measurement tools are supported in conjunction with the volume rendering functions.	Chapter 9
Tiling/Modeling	Functions to create and extract tiled surfaces from segmented volumes and object maps. The tiled surfaces can be saved in a variety of common CAD formats.	Chapter 13

CHAPTER 2

AVW Concepts

The **AVW** library of functions is designed to handle image and volume data in a fast and efficient manner while providing the functions useful for image visualization, processing, and measurement. An image is a 2D array of values called pixels which are ordered in lines, and a volume is a 3D array of values called voxels ordered in images. In this design, multiple types of volumes, referred to as bands, spectra or time points, are represented by an array of volumes. Since in multi-volume data sets each volume has its own structure, multiple bands are not required to have the same dimensions or data type. Image and volume structures have been designed to make function calls simple by requiring fewer parameters. These structures all assume that an image or volume is stored in contiguous memory.

2.1 AVW_Image

Images are represented by the following structure.

FIGURE 2-1 **AVW_Image Structure**

```
typedef struct
    {
    void *Mem;            /* Pointer to the first pixel of the image */
    int DataType;                          /* Identifies the data type */
    unsigned int Width;             /* Number of columns in an image */
    unsigned int Height;               /* Number of rows in an image */
    unsigned int BytesPerPixel;         /* Number of bytes per pixel */
    unsigned int BytesPerLine;              /* BytesPerPixel * Width */
    unsigned int BytesPerImage;    /* BytesPerPixel * Width * Height */
    unsigned int PixelsPerImage;                   /* Width * Height */
    AVW_Colormap *Colormap;         /* Pointer to colormap structure */
    char *Info;            /* Pointer to information character string */
    unsigned int *YTable; /* Array of offsets to each row of an image */
    } AVW_Image;
```

2.1.1 Image Memory

The **Mem** element of the **AVW_Image** structure is a pointer to the array of memory which contains the image data. When **Mem** is used within a routine it is necessary to type cast it to the proper **AVW** data type.

FIGURE 2-2 **Memory Type Casting**

```
AVW_Image *image;
int value;
     .
     .
/* Get the value of the center pixel of the image */

     value = *(((unsigned char *) image->Mem) +
            image->YTable[image->Height/2] + image->Width/2);
```

2.1.2 Data Types

The **DataType** member defines the type of data in the image. The following data types are supported within **AVW**.

FIGURE 2-3 **Data Types From AVW.h**

```
    #define AVW_UNSIGNED_CHAR    (1<<0)    /* 1   */
    #define AVW_SIGNED_CHAR      (1<<1)    /* 2   */
    #define AVW_UNSIGNED_SHORT   (1<<2)    /* 4   */
    #define AVW_SIGNED_SHORT     (1<<3)    /* 8   */
    #define AVW_UNSIGNED_INT     (1<<4)    /* 16  */
    #define AVW_SIGNED_INT       (1<<5)    /* 32  */
    #define AVW_FLOAT            (1<<6)    /* 64  */
    #define AVW_COMPLEX          (1<<7)    /* 128 */
    #define AVW_COLOR            (1<<8)    /* 256 */
```

Most of the data types are self explanatory. Complex pixels are represented by two floating point values, the real and imaginary components. The **`AVW_COLOR`** data type is discussed in the section on Color Images.

2.1.3 Image Dimensions

`Width` is the number of pixels in the X dimension of the image. **`Height`** is the number of pixels in the Y dimension of the image.

2.1.4 Precomputed Values

Many commonly used values have been precomputed and included in the **`AVW_Image`** structure. **`BytesPerPixels`** is the number of bytes needed for one pixel. This value depends on the **`DataType`**. **`BytesPerLine`** is the number of bytes in one row of the image. **`BytesPerImage`** is the number of bytes contained in an entire image. **`PixelsPerImage`** is the number of pixels in an entire image.

2.1.5 Color Lookup Table

`Colormap` contains a color lookup table, which is **`NULL`** for grey scale data. The **`AVW_Colormap`** structure is defined as follows.

FIGURE 2-4 **AVW_Colormap Structure From AVW.h**

```
typedef struct
    {
    int Size;                   /* Size of Red, Green, and Blue Arrays */
    unsigned char *Red;
    unsigned char *Green;
    unsigned char *Blue;
    } AVW_Colormap;
```

2.1.6 Other Information

The **`Info`** string is used to store any relevant information about the image which is not contained in the **`AVW_Image`** structure. The length of this string is increased as information is added to it. Character strings and numeric values may be stored in or retrieved from this string with the following functions: **`AVW_GetNumericInfo`**(), **`AVW_GetStringInfo`**(), **`AVW_PutNumericInfo`**(), and **`AVW_PutStringInfo()`**. The use of these functions is illustrated in the following example.

FIGURE 2-5 **Info String Example**

```
AVW_Image *image;
double ysize;
int id;
char *orient, date[32];
     .
     .
     .
     ysize = AVW_GetNumericInfo("VoxelHeight", image->Info);
     if(AVW_ErrorNumber)
           fprintf(stderr,"VoxelHeight not found. \n");
     else
           printf("Y Size=%f\n", ysize);

     if(!(orient = AVW_GetStringInfo("Orientation", img->Info)))
     {
           fprintf(stderr, "Orientation not found.");
     }
     else
     {
           printf("Orientation=%s\n", orient);
           free(orient);
     }
     image->Info = AVW_PutNumericInfo("ID No.", (double)id, image->Info);
     image->Info = AVW_PutStringInfo("Exp. Date", date, image->Info);
```

2.1.7 Image Offsets

`YTable` is an array of offsets to the start of each row in the image. This allows the user to quickly, by avoiding a multiplication, address pixels within the image. See Figure 2-2.

2.2 AVW_Volume

The **`AVW_Volume`** structure is an extension of the **`AVW_Image`** structure. **`Depth`** defines the number of images in the Z dimension of the volume. **`BytesPerVolume`** and **`VoxelsPerVolume`** provide commonly used pre-computed values. The **`ZTable`** is an array of offsets to the start of each image within the volume. The **`AVW_Volume`** structure is defined as follows.

FIGURE 2-6 **AVW_Volume Structure**

```
typedef struct
    {
    void *Mem;              /* Pointer to the first voxel of the volume */
    int DataType;                               /* Identifies data type */
    unsigned int Width;                    /* Number of columns per image */
    unsigned int Height;                      /* Number of rows per image */
    unsigned int Depth;                    /* Number of images per volume */
    unsigned int BytesPerPixel               /* Number of bytes per pixel */
    unsigned int BytesPerLine;                   /* BytesPerPixel * Width */
    unsigned int BytesPerImage;       /* BytesPerPixel * Width * Height */
    unsigned int PixelsPerImage;                        /* Width * Height */
    unsigned int BytesPerVolume;                 /* BytesPerImage * Depth */
    unsigned int VoxelsPerVolume;               /* Width * Height * Depth */
    AVW_Colormap *Colormap;             /* Pointer to colormap structure */
    char *Info;             /* Pointer to information character string */
    unsigned int *ZTable;        /* Offsets to each image in the volume */
    unsigned int *YTable;   /* Array of offsets to each row of an image */
    } AVW_Volume;
```

2.3 Create and Destroy

An **`AVW_Image`** or **`AVW_Volume`** can easily be created or destroyed with provided **`AVW`** functions. These functions will allocate memory based on user specified parameters or will use an existing block of memory. The following example demonstrates: the creation of an **`AVW_Volume`** where the memory is allocated by **`AVW_CreateVolume()`**, the creation of an **`AVW_Image`** where memory has been allocated prior to the call to **`AVW_CreateImage()`**, the destruction of an **`AVW_Volume`** and all its associated memory, and the destruction of an **`AVW_Image`** with preservation of the image memory. The demonstrated creation of the **`AVW_Image`** is shown for illustrative purposes and does not imply a recommended method unless the allocation of memory is done by some other routine than malloc().

FIGURE 2-7 **Create and Destroy Example**

```
AVW_Image *image;
AVW_Volume *volume;
int width=256, height=300, depth=50, type, size;
unsigned char *data;
     .
     .
     .
     type = AVW_UNSIGNED_CHAR;
     volume = AVW_CreateVolume(NULL, width, height, depth, type);

     size = width * height * sizeof(unsigned char);
     data = (unsigned char *) malloc(size);

     image = AVW_CreateImage(data, width, height, AVW_UNSIGNED_CHAR);
     .
     .
     image->Mem = NULL;     /* Prevent the destroy from freeing memory */
     AVW_DestroyImage(image);
     image = NULL;

     AVW_DestroyVolume(volume);
     volume = NULL;
     free(data);
```

The macros **`AVW_DESTROYIMAGE()`** and **`AVW_DESTROYVOLUME()`** are defined in **`AVW.h`**. These macros perform the same functions as **`AVW_DestroyImage()`** and **`AVW_DestroyVolume()`** and also set the image or volume to **`NULL`**. Debugging may become more difficult when using these macros rather than the functions.

2.4 Memory Usage

AVW provides a mechanism for reusing and allocating **AVW** structures. This method removes much of the burden of memory allocation and destruction from the programmer. It also facilitates the efficient reuse of images in loops. Most **AVW** functions store intermediate results internally when necessary thus allowing the same image or volume to be used as both an input and output parameter. This includes routines which may change the size or data type of the image or volume. Though this section will only discuss **AVW_Images**, the same rules apply to **AVW_Volumes** and other **AVW** structures.

Many **AVW** functions return an **AVW_Image** and also have an out_image as part of the function's parameter list.

```
     AVW_Image *AVW_Function(in_image, out_image)
```

The parameter **out_image** is provided as a method of reusing an existing **AVW_Image**. Reuse is possible if, and only if, the size and data type of the provided **out_image** meet the requirements of the function. In this case the pointer to **out_image** is returned by the function. If not reusable and not **NULL**, **out_image** will be released and reallocated.

A **NULL** passed in as the output image parameter guarantees the creation of a satisfactory **AVW_Image** which is returned by the function. It is often a good practice to initialize **AVW_Image** pointers to **NULL** upon declaration.

Two typical ways of using this capability through the function calls are:

FIGURE 2-8 **Image Recycling**

```
AVW_Image *in_image, *out_image1 = NULL, *out_image2;
     .
     .
     .

     out_image1 = AVW_CopyImage(in_image, out_image1);
     out_image2 = AVW_CopyImage(in_image, NULL);
```

In the first call **out_image1** may be reused (if it was used previously) and in the second call **out_image2** is created. In any case, the returned pointer should always be used as the results of the function. Note that in order to use the first call **out_image1** should be declared as follows:

```
     AVW_Image *out_image1 = NULL;
```

Any output image will be freed if an error occurs in the function and **NULL** will be returned.

In the following example, **`image`** appears as both the returned pointer and as an input parameter. The first time **`AVW_GetOrthogonal()`** is called, **`image`** will be **`NULL`**, and **`AVW_GetOrthogonal()`** will call **`AVW_CreateImage()`** to allocate the required **`AVW_Image`**. On successive calls to **`AVW_GetOrthogonal()`**, **`image`** will contain a valid and reusable **`AVW_Image`**.

FIGURE 2-9 Image Reuse Example

```
#include "AVW.h"

int test(volume)
AVW_Volume *volume;
{
    AVW_Image *image = NULL;
    int i;
    for(i=0;i<volume->Depth;++i)
    {
          image = AVW_GetOrthogonal(volume, AVW_TRANSVERSE, i, image);
          if(!image)
          {
                AVW_Error("AVW_GetOrthogonal");
                return(AVW_FAIL);
          }
          YOUR_DISPLAY_FUNCTION(image);
    }
    AVW_DestroyImage(image);
    return(AVW_SUCCESS);
}
```

2.5 Color Images

There are two types of color images supported within **`AVW`**: **`AVW_UNSIGNED_CHAR`** with an **`AVW_Colormap`** and **`AVW_COLOR`** (24-bit RGB). **`AVW_COLOR`** images are represented by three consecutive bands which correspond to the red, green, and blue components of the color image. Each value is an unsigned char. The **`Mem`** element of the image structure points to the first value of the red band. This is followed by the rest of the red band values, which are followed by the green and blue band values. Each of the pre-computed values within the **`AVW_Image`** structure, such as **`PixelsPerImage`**, refer to a single band.

2.6 Points and Point Lists

`AVW` has structures which define many types of 2D and 3D points. These structures allow for concise and consistent parameter passing within the **`AVW`** functions. A variety of point structures are available to hold a number of different coordinate data types including: short int, int and float.

FIGURE 2-10 **Point Structures**

```
typedef struct
{
    short X, Y;
} AVW_Point2;

typedef struct
{
    int X, Y;
} AVW_IPoint2;

typedef struct
{
    float X, Y;
} AVW_FPoint2;

typedef struct
{
    short X, Y, Z;
    short padding;
} AVW_Point3;

typedef struct
{
    int X, Y, Z;
} AVW_IPoint3;

typedef struct
{
    float X, Y, Z;
} AVW_FPoint3;
```

These are often combined in other structures to describe lines and rectangles.

AVW also provides structures for self-managed lists of points which can be used for such things as traces and stacks.

FIGURE 2-11 Point List Structure

```
typedef struct
{
    unsigned int NumberOfPoints;
    unsigned int MaximumPoints;
    unsigned int BlockSize;
    AVW_Point2 *Points;
} AVW_PointList2;
```

NumberOfPoints specifies the current number of points in the list. **MaximumPoints** specifies the maximum number of points allowed in the list before the automatic reallocation of memory occurs. **BlockSize** specifies the number of points the list is initialized to and increased by each time a reallocation of memory occurs. **Points** is an array of **AVW_Point2** structures. Similar point list structures exist for many of the other types of defined points.

Functions for creating, destroying and adding or removing a point to the lists also exist.

FIGURE 2-12 Point List Example

```
AVW_PointList2 *list;
AVW_Point2 point;

list = AVW_CreatePointList2(256);
```

```
for(i=0; i < 256; i++)
{
     point.X = i;
     point.Y = i;
     AVW_AddPoint2(list. &point);
}

AVW_RemovePoint2(list, list->NumberOfPoints/2);/* Remove middle point */

for(i=0; i < list->NumberOfPoints; i++)
{
     printf(“%d : %d, %d\n”,i, list->Points[i].X, list->Points[i].Y);
}
AVW_DestroyPointList2(list);
list = NULL;
```

2.7 Volume Coordinate System

Most **AVW** functions are coordinate system independent. Functions that are not, assume a left-handed coordinate system with the origin in the front lower left corner. A major advantage of this convention is that the coordinate origin of each orthogonal orientation (transverse, coronal, sagittal) lies in the lower left corner of the extracted image. If the display coordinate system does not match this convention the images may need to be flipped prior to display.

2.8 Function Specification and Calling Conventions

The **AVW** library function names start with **AVW_** followed by the name of the function which has the first letter of each word capitalized. The parameters for the functions follow the pattern:

```
     AVW_FunctionName(input parameter(s), ..., output parameter(s))
```

2.9 Error Handling

All of the **AVW** functions support error checking in a consistent manner, providing the user with an error number and an error message if a function fails. See the **AVW_Errors.h** include file for a detailed list of each error code which may be returned.

2.10 Man Pages

The **AVW Reference Manual,** which includes documentation of **AVW** functions, structures and macros, exists online. Setting of the MANPATH to include **$AVW/man** allows all of the documentation to be viewed using the UNIX “man” command.

2.11 ANSI C Prototyping

The prototypes for each **AVW** function are defined in the **AVW** include files. If a non-ANSI compiler must be used, **K_R** must defined to disable prototype checking. If an ANSI compiler is used, conflicts of parameters will be reported.

2.12 Compiling and Linking

The following simple example demonstrates the compile and link of a program containing **AVW** function calls.

```
cc test.c -o test -I$AVW/include -L$AVW/$TARGET/lib -lAVW -lm
```

This example assumes that the **AVW** environment variable has been set to the directory where **AVW** was installed. The **TARGET** environment variable must be set to the target system name (SPARC, SGI5, HP, ALPHA).

CHAPTER 3

Image I/O

The **AVW** image I/O functions are designed to make it easy to read and write image files from various file formats using a single set of functions. This eliminates the need to convert images into a common format creating redundant data. Image data can be easily converted between various formats and ported to other applications. Facilities are provided for the user to extend the I/O functions to read and write other formats.

AVW image I/O functions automatically recognize common formats. The identical code can be used to read a *GE Advantage CT* file or a *Sun Raster* file. A single parameter controls whether an image is written to disk as *PostScript* or *PPM*. New image file formats are continually being added and a facility is also provided to enable users to extend the **AVW** image I/O functions to support additional file formats as well. It is also easy to read "raw" data into an **AVW_Image** structure.

3.1 Similarity to UNIX File I/O

AVW image file I/O is similar to standard file I/O in UNIX. Just as UNIX functions open files and read and write blocks of data, **AVW** functions open recognized image files and read and write images.

• **AVW_OpenImageFile()**	opens an image file.; similar fopen()
• **AVW_ReadImageFile()**	reads an image from and image file; similar to fread()
• **AVW_SeekImageFile()**	position to a specified image; similar to fseek()
• **AVW_WriteImageFile()**	write an image to a file; similar to fwrite()
• **AVW_CloseImageFile()**	closes an image file; similar to fclose()
• **AVW_CreateImageFile()**	creates an image file; similar to create() or fopen(filename,”w”);
• **AVW_ExtendImageFile()**	extends support for a additional image file format.

There are also functions for extending support for other file formats. Image files (even those containing a single image) are considered to be four dimensional. **Analyze** and AVW image files may contain multiple volumes of multiple slices. A single PPM file is treated as if it has one volume and one slice.

3.2 The Data Structures

3.2.1 AVW_ImageFile

The **`AVW_ImageFile`** structure contains generic information about any image file successfully opened by **`AVW`**. This structure is returned by **`AVW_OpenImageFile()`** and **`AVW_CreateImageFile()`** in the same way that a pointer to **`FILE`** is returned by **`fopen().`** It is used as an argument to the other **`AVW Image I/O`** functions. This structure is found in **`AVW_ImageFile.h.`**

FIGURE 3-1 **The AVW_ImageFile Structure**

```
typedef struct
{
    char *FileName;                                  /* name of file */
    char *FileModes;                                   /* open modes */
    int DataFormat;        /* index in the list of supported formats */
    int DataType;                         /* AVW_UNSIGNED_CHAR, etc. */
    unsigned int Width;                /* number of pixels in a row */
    unsigned int Height;              /* number of rows in an image */
    unsigned int Depth;             /* number of slices in a volume */
    unsigned int NumVols;              /* number of volumes in file */
    unsigned int BitsPerPixel;       /* useful derived fields  */
    unsigned int BytesPerPixel;
    unsigned int BytesPerLine;
    unsigned int BytesPerImage;
    unsigned int BytesPerVolume;
    unsigned int BytesPerFile;
    unsigned int PixelsPerImage;
    unsigned int VoxelsPerVolume;
    unsigned int VoxelsPerFile;
    int CurrentSlice, CurrentVolume;        /* current file position */
    AVW_Colormap *Colormap;
    void *NativeData;              /* points to format specific data */
    char *Info;
} AVW_ImageFile;
```

The elements of this structure should be considered as "read only" by the developer. The example program **`showatt.c`** in **`$AVW/extras`** opens an image file and displays the information contained in this structure.

```
AVW_ImageFile *inf;
             ;
      if((inf = AVW_OpenImageFile(filename,”r”)) == AVW_NULL)
       {
           AVW_Error(filename);
           return
       }
       printf(“File = %s\n”,inf->FileName);
       printf(“  DataType   = %3d\n”, inf->DataType);
       printf(“  Width   = %3d\n”,inf->Width);
       printf(“  Height  = %3d\n”,inf->Height);
       printf(“  Depth   = %3d\n”,inf->Depth);
       printf(“  NumVols = %3d\n”,inf->NumVols);
               ;
```

3.2.2 Extending to Other Formats

The **`AVW_ExtendIO`** structure is used to group the functions and information necessary for **`AVW`** to support a new image file format. **`AVW`** maintains an entry of this structure for each supported image file format. **`AVW_ListFormats()`** can be used to get a list of currently extended image file formats which fulfill a set of criteria. See example program **`listformats.c`**. To extend **`AVW`** image I/O to support another image file format elements of this structure must be set to appropriate values and functions and then the structure is passed to **`AVW_ExtendIO()`**.

FIGURE 3-2 **The AVW_ExtendIO structure**

```
typedef struct
    {
    char             *Extension;   /* extension if any for format */
    char             *Description; /* name by which format is known */
   int             MagicNumber;/* optional magic number  */
   int               Properties; /* supported properties flag */
```

```
  AVW_ImageFile * (*Open)();   /* the open function */
  int             (*Seek)();   /* the seek function */
  AVW_Image *     (*Read)();   /* the image read function */
  int             (*Write)();  /* the image write function */
  int             (*Close)();  /* file close function */
  AVW_ImageFile * (*Create)(); /* file creation function */
  int             (*Query)();  /* identification funct*/
} AVW_ExtendIO;
```

3.3 AVW Image I/O Functions

These functions support image file I/O in **AVW**.

FIGURE 3-3 AVW I/O Functions

Reading and Writing

`AVW_OpenImageFile()`	opens an image file
`AVW_SeekImageFile()`	seeks to an image in an image file
`AVW_ReadImageFile()`	reads an image from a file
`AVW_ReadVolume()`	reads a volume from a file
`AVW_WriteImageFile()`	writes an image to a file
`AVW_WriteVolume()`	writes a volume to a file
`AVW_CloseImageFile()`	closes an image file
`AVW_CreateImageFile()`	creates an image file (for writing)

File format support

`AVW_ExtendImageFile()`	extends support for another file format
`AVW_FormatSupports()`	determines a file format supports property
`AVW_ListFormats()`	lists file formats supporting properties
`AVW_ExtendExternalLibs()`	loads additional shared libraries
`AVW_DisableImageFileFormat()`	disables a file format
`AVW_EnableImageFileFormat()`	undoes `AVW_DisableImageFileFormat()`

3.4 Properties

Image file formats have different properties associated with them. These properties have to do with the voxel data types supported, whether the format supports 3D and 4D, and whether AVW has been enabled to read and write this format.

Some formats can store image data in many data types and some only one. Some can store only a single image and others are capable of storing multiple 3D volume data sets. For example CT and MR scanner formats typically support one image per file with 16 bit **`(AVW_SIGNED_SHORT)`** pixels. Interfile format image files can consist of multiple 3D image sets, and the pixels can be of several data types. Additionally some formats are supported with only read capability; for example GE Advantage image files may be read but not written. When a format is extended for **`AVW`** the **`Properties`** element of the **`AVW_ExtendIO`** structure is updated by combining the following mask values found in **`AVW_ImageFile.h`**.

FIGURE 3-4 **Image File Format Properties**

```
Supported Data Types
    AVW_SUPPORT_UNSIGNED_CHAR
    AVW_SUPPORT_SIGNED_CHAR
    AVW_SUPPORT_UNSIGNED_SHORT
    AVW_SUPPORT_SIGNED_SHORT
    AVW_SUPPORT_UNSIGNED_INT
    AVW_SUPPORT_SIGNED_INT
    AVW_SUPPORT_FLOAT
    AVW_SUPPORT_COMPLEX
    AVW_SUPPORT_COLOR

Supported Dimensions
    AVW_SUPPORT_2D
    AVW_SUPPORT_3D
    AVW_SUPPORT_4D

Read/Write
    AVW_SUPPORT_READ
    AVW_SUPPORT_WRITE
```

For example, **AVW** supports the reading and not the writing of GE Advantage format image files. The property value for the GE Advantage format which **AVW** supports for reading only is:

```
    AVW_SUPPORT_SIGNED_SHORT|AVW_SUPPORT_2D|AVW_SUPPORT_READ
```

Attempts to create a GE Advantage file in **AVW** will fail because this format was implemented as read only. Likewise attempting to write an image of data type **`AVW_COMPLEX`** to a PPM file will fail because this format does not support this data type. A properties value is also passed to **`AVW_ListFormats()`** to get a list of image file formats fulfilling a specified criteria.

FIGURE 3-5 **Listing File Format Names by Supported Properties**

```
AVW_List *list;
int properties;

properties = AVW_SUPPORT_READ|AVW_SUPPORT_WRITE|AVW_SUPPORT_FLOAT;
list = AVW_ListFormats(properties);
```

The returned **`list`** will contain all currently configured formats which support reading and writing of floating point images.

`AVW_FormatSupports()` is used to determine if a format supports a property or group of properties. For example to determine if floating point images can be written to a format **`ff`**;

FIGURE 3-6 **Determining if a File Format Accepts a Floating Point Image**

```
char *ff;
.
.
.
if(AVW_FormatSupports(ff, AVW_SUPPORTS_FLOAT|AVW_SUPPORTS_WRITE))
{
    /* yes */
}
```

3.5 AVW Info Strings

Additional information from image files which may or may not be present in all types of files may be returned in the **`Info`** strings of the **`AVW_ImageFile`** structure after opening an image file or in the **`AVW_Image`** structure after reading and image. Some standard **`Info`** tags set when appropriate are: "*VoxelWidth*", "*VoxelHeight*", "*VoxelDepth*", "*SliceLocation*", *"Patient Name"*, *"Orientation"*, *"Exam Number"*, *"Series Number"*, *"Image Number"*. Info tags that begin with a period (.) are useed internally by AVW. These tags may not be present in all image file formats. If they are not present To determine if these tags are present use **`AVW_GetNumericInfo()`**. **`Info`** strings are unique to **AVW** and not supported by other image formats. The previously mentioned tags are used to set voxel dimensions when writing **`AnalyzeImage`** format files. The two **AVW** image file formats **`AVW_VolumeFile`** and **`AVW_ImageFile`** do write all **Info** string entries to the file.

Formats other than **`AVW_ImageFile`** have no way to hold **`Info`** string data. For example additional Info string items are not written to disk when storing a file in **`AnalyzeImage`** or **`SunRaster`** formats. The only format which preserves Info string entries is the **`AVW_ImageFile`** (see section 3.9.2).

To set the voxel dimension header fields when writing images to an **`AnalyzeImage`** format file the information must be put into the **`Info`** string of the **`AVW_ImageFile`** structure before writing images to the file, for example:

FIGURE 3-7 Setting the Voxel Dimensions of an Image File

```
AVW_ImageFile *imgfl;
AVW_Image *img;
double voxwidth, voxheight, voxdepth;
.
.  /* imgfile is created , voxel dimensions are set, img is created */
.
imgfl->Info = AVW_PutNumericInfo("VoxelWidth",voxwidth,imgfil->Info);
imgfl->Info = AVW_PutNumericInfo("VoxelHeight",voxheight,imgfil->Info);
imgfl->Info = AVW_PutNumericInfo("VoxelDepth", voxdepth, imgfil->Info);

AVW_WriteImageFile(imgfl,img);

AVW_CloseImageFile(imgfl);
```

When Interfile format images are opened with **`AVW_OpenImageFile()`** every Interfile tag in the file is stored in the **`Info`** string.

FIGURE 3-8 **Info String Entries From an Interfile Format Image File**

```
"!number format"="UNSIGNED INTEGER"
"!number of bytes per pixel"="2"
"!type of data"="STATIC"
```

3.6 Image File Formats

AVW supports the following image file formats.

FIGURE 3-9 **Supported AVW Image File Formats**

```
AVW_ImageFile
AVW_VolumeFile
AnalyzeImage
AnalyzeScreen
BMP
CTI           (read only)
GE9800        (read only)
GE Advantage  (read only)
GESigna       (read only)
GESTARCAM
SunRaster
Imatron       (read only)
SiemensCT     (read only)
INTERFILE     (read only)
ACRNEMA       (read only)
PAPYRUS       (read only)
SGIrgb
SGIbw
PPM
```

```
    PGM
    YUV
    PIC
    PICKERMRI     (read only)
    SMIS          (read only)
    XBM
    Postscript    (write only)
    TARGA
    JPG
```

Note that all formats regardless of their individual properties are treated by **AVW** as being 3D. Single slice file formats are treated as containing a single volume of one slice. In this way there is a single API to all formats.

Additional tools may be provided to extend support for PAPYRUS3, TIFF, and DICOM via external extended libraries.

3.7 Special AVW Image File Formats

AVW provides two file formats specific to **AVW**: **`AVW_VolumeFile`** and **`AVW_ImageFile`**. Both of these formats support the **AVW `Info`** string which enables applications to attach any type of numeric or string information to an image or volume.

3.7.1 AVW_VolumeFile Format

The **`AVW_VolumeFile`** format provides a way of logically grouping a series of single slice image files into a single 3D entity which can be opened and read from as if it were a single 3D file. CT scanner formats typically store each slice in a 3D acquisition as a single file. To create an **`AVW_VolumeFile`**; create a text file and put "*AVW_VolumeFile*" as the first line of text. Each subsequent line in the file contains the name of an image file (in order) of the 3D data set.

FIGURE 3-10 **Example of an AVW_VolumeFile**

```
AVW_VolumeFile
ST507SER001.001
ST507SER002.001
... etc.
```

If a directory contains only image files of the same size and data type whose file names sort alphabetically to anatomical order, the name of the directory may be passed as a filename to **`AVW_OpenImageFile()`**. A wild card string that matches files in a directory known to be part of a 3D set can also be passed to **`AVW_OpenImageFile()`**.

FIGURE 3-11 **Opening an AVW_VolumeFile with Wild Card File Specification**

```
AVW_Imagefile *imgfl;

imgfl = AVW_OpenImageFile("ST507SSER001.*", "r");
```

Although the **`AVW_VolumeFile`** format is intended as a way to group existing one slice files of any format into a 3D entity, image files can be created in this format as well. When individual slices are written in this format, the name is built using the name of the **`AVW_VolumeFile`** text file and the slice number. The **`DataFormat`** of the individual slice files is assigned unless **`SecondaryFileFormat`** is specified in the **`Info`** string. The following example shows how to write all images from a volume to an **`AVW_VolumeFile`** as individual SunRaster files.

FIGURE 3-12 **Writing Images to an `AVW_VolumeFile` and Individual SunRaster Files**

```
AVW_Image  *img=NULL;
AVW_Volume *vol;
AVW_ImageFile *imgfl;
char *outname;
int i;
.
.
.
outfile = AVW_CreateImageFile("Test","AVW_VolumeFile, vol->Width,
                                vol->Height, vol->Depth, vol->DataType);
if(!outfile)
{
    /*error handling */
}
outfile->Info = AVW_PutStringInfo("SecondaryFileFormat", "SunRaster",
                                                          outfile->Info);
.
for(i=0;i<vol->Depth;i++)
{
    img = AVW_GetOrthogonal(vol, AVW_TRANSVERSE, i, img);
    AVW_SeekImageFile(outfile, 0, i);
    AVW_WriteImageFile(outfile, img);
}
```

The **AVW_VolumeFile** format can also be used to store descriptions of subvolumes. **AVW_WriteSubVolumeDescription()** creates a file in the **AVW_VolumeFile** format which points to another image file of any supported format. Once created; opening and reading from this file returns the specified subvolume from the **AVW_SubvolSourceFile**. Creation of subvolume description files provides a mechanism for easy definition of and retrieval of subvolumes and/or regions from image files without copying the voxel data to a new file.

FIGURE 3-13 Creating a SubVolume Description File

```
char outname[256];
char infile[256];
int involnum = 0;
AVW_Rect3 subv;

subv.PointA.X = 32;
subv.PointA.Y = 30;
subv.PointA.Z = 25;

subv.PointB.X = 65;
subv.PointA.Y = 60;
subv.PointA.Z = 50;
strcpy(outname, “dsrdog_subvol”);
strcpy(infile, “dsrdog.hdr”);
AVW_WriteSubVolumeDescription(outname, infile, involnum, &subv, “ “);
```

Results in File “dsrdog_subvol”:

```
AVW_VolumeFile
```

```
#AVW_SubVolumeDescription
#AVW_SubVolumeSourceFile = dsrdog.hdr
#AVW_SubVolumeInputVolumeNumber = 0
#AVW_SubVolumePointA = 32   30   25
#AVW_SubVolumePointB = 65   60   50
```

This subvolume description file points to the **AnalyzeImage** format file *dsrdog.hdr.* The subvolume description encompasses columns 32 through 65 and rows 30 through 60 of slices 25 through 50 of volume number 0. Reading image number 0 from this file will return the specified region from slice 25 of *dsrdog*.

3.7.2 AVW_ImageFile Format

The **`AVW_ImageFile`** format supports and maintains **AVW `Info`** strings, multiple 3D image sets, and all data types. The header and voxel data are in the same file and the header fields are text lines of the form:

```
Tag=Value
```

making it easy to view file attributes with text based utilities. The first line in the file of an **`AVW_ImageFile`** contains the tag **`AVW_ImageFile`**, a version number, and byte offset to image data. The rest is self-explanatory.

FIGURE 3-14 **Text Contents of an AVW_ImageFile**

```
AVW_ImageFile   1.00     8192
DataType=AVW_SIGNED_SHORT
Width=256
Height=256
Depth=3
NumVols=1
ColormapSize=0
BeginInformation
MaximumDataValue=2553
MinimumDataValue=3
EndInformation
MoreInformation=-1
Vol Slc  Offset    Length  Cmp Format
  0   0     8192   131072   0
  0   1   139264   131072   0
  0   2   270336   131072   0
EndSliceTable
```

The pixel information is may or may not be compressed. Non-compressed or contiguous files are used as memory mapped files for loaded volumes in AnalyzeAVW. This file format supports 3D, 4D, all data types, and the **AVW Info** string. While the header information is in ASCII text, the file should not be modified with a text editor. Future development will include support for various compression schemes.

3.8 Reading Raw Data into an AVW_Image

Image data contained in unsupported formats can be easily read into **AVW_Image** structures if the image data is contiguous, uncompressed, and of an **AVW** supported data type. For example if **fp** is a file handle to an open file which contains pixel data of known attributes it may be read into an **`AVW_Image`** as follows:

FIGURE 3-15 Reading Data into an AVW_Image

```
FILE *fp;
AVW_Image img;
int width, height, type, offset;
.
.        /* set width, height, type, and offset to appropriate values */
.
img = AVW_CreateImage(AVW_NULL, width, height, type);
if(!img)
{
     /* handle error */
}
else
{
     fseek(fp, offset, o);
     fread(img->Mem, 1, img->BytesPerImage, fp);
}
```

Other **`AVW`** functions which may be useful for reading and manipulating raw data are:
   `AVW_SwapBlock(), AVW_ReadSwap(), AVW_ReverseBits(),`

`AVW_SwapDouble(), AVW_SwapFloat(), AVW_SwapInt(), AVW_SwapLong(),AVW_SwapShort(), AVW_WriteSwap()` and **`AVW_ReadSwap().`**

Additionally any such file may be opened as if it is an image file by by setting the file mode to UNKNOWN in the call to AVW_OpenImageFile. The elements of the AVW_ImageFile structure are set to appropriate values before invoking AVW_ReadImageFile(). Additional processing instructions may be put in the **Info** string of the **AVW_ImageFile** structure.

FIGURE 3-16 **Opening and reading images from an UNKNOWN file:**

```
AVW_ImageFile *imgfl=NULL;
AVW_Image *img=NULL;
int width, height, depth, datatype, voxelOffset;
/* set width, height, depth, & datatype */
imgfl = AVW_OpenImageFile(filename, “UNKNOWN”);
imgfl->Width    = width;
imgfl->Height   = height;
imgfl->Depth    = depth;
imgfl->DataType = datatype;
     /* specify byte offset at which pixel data starts */
imgfl->Info =
    AVW_PutNumericInfo(“VoxelOffset”, voxelOffset, imgfl->Info);
   /* None, Pairs, Quads */
imgfl->Info = AVW_PutStringInfo(“ByteSwap”, “Pairs”, imgfl->Info);
   /* Yes, No */
imgfl->Info = AVW_PutStringInfo(“ReverseBits”, “No”, imgfl->Info);
   /* Yes, No */
imgfl->Info = AVW_PutStringInfo(“FlipX”, “No”, imgfl->Info);
   /* Yes, No */
imgfl->Info = AVW_PutStringInfo(“FlipY”, “Yes”, imgfl->Info);
ret = AVW_SeekImageFile (imgfl, volnum, slicnum);
img = AVW_ReadImageFile(imgfl, img);
```

;

3.9 Extending I/O for Other Image File Formats

Additional files in the **`$AVW/extras`** directory provide support for *TIFF*

and *DICOM* by linking to additional software libraries that are readily accessible via anonymous ftp. These are provided as examples of how to write functions which can be passed to **`AVW_ExtendIO()`** to link **AVW** I/O functions with other software libraries.

`AVW` I/O functions can be extended to support additional image file formats. The developer needs to supply functions which interface to a exiting functions which handle the new format. Examples of how this is done for *TIFF* and *DICOM* are in the **`$AVW/extras/io/expand_io`**. Basically what is involved is the writing of six functions which interface between the AVW functions and the format specific functions. These functions take the same arguments and return the same types as their corresponding **`AVW`** image I/O functions.

FIGURE 3-17 **I/O Extension Functions**

query()	returns 1 or 0 if the file is of this format
open()	returns an AVW_ImageFile structure for this file
read()	returns an AVW_Image
write()	writes an AVW_Image
close()	closes file and releases memory specific to this format
create()	returns an AVW_ImageFile structure ready for writing

FIGURE 3-18 **Extending AVW to Support TIFFAVW_ImageFile *avw_open_tiff_image();**

```
int            avw_seek_tiff_image();
AVW_Image      *avw_read_tiff_image();
int            avw_write_tiff_image();
int            avw_close_tiff_image();
AVW_ImageFile  *avw_create_tiff_image();
int            avw_query_tiff_image();

int avw_ExtendForTIFF()
{
int val;
AVW_ImageFile_Functions *fl;

if((fl =(AVW_ExtendIO*)mal-
loc(sizeof(AVW_ImageFile_Functions)))==AVW_NULL)
     return(AVW_FAIL);

fl->Extension =  ".tif";
fl->Description = "TIFF";   /* Tag by which AVW will know this format */
fl->MagicNumber = 0;                /* Actually TIFF has 2; II and MM */

fl->Properties = AVW_SUPPORT_UNSIGNED_CHAR | AVW_SUPPORT_COLOR |
                                   AVW_SUPPORT_READ | AVW_SUPPORT_WRITE;

fl->Open = avw_open_tiff_image;
fl->Seek = avw_seek_tiff_image;
fl->Read = avw_read_tiff_image;
fl->Write = avw_write_tiff_image;
fl->Close = avw_close_tiff_image;
fl->Create = avw_create_tiff_image;
fl->Query = avw_query_tiff_image;               /* Is it a TIFF file? */

val = AVW_ExtendImageFile(fl);
free(fl);
return(val);
}
```

The **`avw_open_tiff_image()`**, **`avw_seek_tiff_image()`**, **`avw_read_tiff_image()`**, **`avw_write_tiff_image()`**, **`avw_close_tiff_image()`**, **`avw_create_tiff_image()`**, and **`avw_query_tiff_image()`** routines are all supplied by the user, and act as an interface between the **`AVW Image I/O`** routines and the functions in the TIFF library.

Once a format specific library is created for a given format it may be more convenient to extend for this format at runtime by loading a shared library under the control of a configuration file. The function call AVW_ExtendExternaLibs() loads the shared libraries specified in the **EXTEND.conf** file in the user's app-defaults directory, AVW's app-defaults directory, or a file specified by the environment variable AVW_Extend. External libraries for support of DICOM, PAPYRUS3, and GIF image file formats may be provided with the AVW distribution.

3.10 Example Programs

Several example programs demonstrating **AVW** image I/O are included in the **$AVW/extras** directory.

FIGURE 3-19 **Example I/O Programs**

Program	**Description**
`showatt.c`	This program calls **AVW_OpenImageFile()** for file names given as command line arguments and reports the values of all elements of the **AVW_ImageFile** structure.
`listformats.c`	Demonstrates use of **AVW_ListFormats()** to get and displaylists of image file format names with specified properties. It shows how to get a list of image file format names which support read, write, read and write, and other various attributes.
`cvrtimg.c`	Reads an image specified by volume and slice number from a file, creates another image file of specified format and writes that image to it.

Program	Description
`cvrtvol.c`	Reads a specified volume from a file, and writes it to another image file of specified file format. Reads and writes volume at a time
`cvrtvol1.c`	Same as cvrtol.c except reads and writes an image at a time
`cvrtvol2.c`	Same as cvrtvol1.c except output file is appended to image at a time.
`cvrttiff.c`	Same as cvrtimg.c with TIFF support.
`cvrtDICOM.c`	Same as cvrtimg.c with DICOM support.

3.11 TCL-TK Implementation

All AVW Image IO routines except AVW_ExtendImageFile() have been implemented in tcl. Applications written in tcl which require extension of user formats should use EXTEND.conf configuration file tp specify loading of shared libraries at run time.

Most of the above ‘C’ sample programs have a corresponding .tcl file which demonstrates how to do the same thing in tcl.

FIGURE 3-20 tcl Image I/O example code

```
# open an image file
    if [catch {set imgf [AVW_OpenImageFile $iname “r”]}] \
    {
      AVW_Error “<Error opening:<$iname>  “
      return
    }
#   Display some information about the file
    puts “File = $iname”
    puts “  DataFormat = [$imgf DataFormat]  “
    puts “  DataType   = [$imgf DataType]”

    puts “  Width     =  [$imgf Width]”
    puts “  Height    =  [$imgf Height]”
    puts “  Depth     =  [$imgf Depth]”
    puts “  NumVols   =  [$imgf NumVols] “


# Display contents of the Info string
    set info [$imgf Info]
    set linfo [AVW_ListInfo $info]
    set ne [$linfo NumberOfEntries]
    for {set i 0} {$i < $ne} {incr i} \
    {
      puts “[$linfo Entry $i]”
    }
    AVW_DestroyList $linfo


# Read an image from the file
set img NULL
set img [AVW_ReadImageFIle $imgfl $img]
AVW_CloseImageFile $imgfl
```

CHAPTER 4

Transform Functions

The Transform Functions encompass a wide variety of routines which perform basic manipulations of images and volumes. These basic operations are often a prerequisite to performing other operations. This chapter will discuss: extracting images from a volume, copying, setting, flipping and shifting of images and volumes, converting images and volumes to different data types, padding, subregioning and resizing of images and volumes, applying mathematical operations to images and volumes, and transforming an image or volume with a matrix.

4.1 Plane Extraction

It is often desirable to extract 2D planes from a volume. **`AVW`** provides routines to get orthogonal, oblique, and curved planes. **`AVW`** functions are also available to put orthogonal or oblique planes into a volume.

4.1.1 Orthogonal Planes

Orthogonal planes are those planes which are parallel to the sides of the input volume. Any particular plane can be described by an orientation and slice number.

`AVW.h` contains three pre-defined values to specify orientation.

FIGURE 4-1 **Excerpts from AVW.h**

```
#define AVW_TRANSVERSE      0
#define AVW_CORONAL         1
#define AVW_SAGITTAL        2
```

The range of legal slice values depends on the dimensions of the input volume. Transverse slices are numbered zero to the depth of the volume minus one. Coronal and Sagittal slices are numbered zero to the height or width of the volume respectively.

The example code in Figure 4-2 shows how **`AVW_GetOrthognal()`** is used to extract an orthogonal slice from a volume. If successful, an **`AVW_Image`** is returned. After processing, it's common for the image to be put either back into the original volume or to another volume with **`AVW_PutOrthogonal()`**.

FIGURE 4-2 **Orthogonal Plane Example**

```
AVW_Volume *volume;
AVW_Image *image;
int slice = 0;
int orient = AVW_TRANSVERSE;
  .
  .
  .
image = AVW_GetOrthogonal(volume, orient, slice, NULL);
if(!image) handle_error();
  .
  .
  .
if(!AVW_PutOrthogonal(image, volume, orient, slice)) handle_error();
  .
  .
  .
```

4.1.2 Oblique Planes

Often the structure of interest is best viewed in an arbitrarily oriented plane. This plane can be described by a series of translations and rotations which can be passed to the **`AVW`** oblique sectioning routine in the form of a matrix.

The oblique image is generated from the input volume, by applying the matrix. An identity matrix would generate a slice equivalent to the center transverse slice. The dimensions of the oblique image are determined either by the size of the **`out_image`** passed to the routine or, if a **`NULL`** **`out_image`** is passed, by the largest axis within the volume.

A choice of interpolation is provided on the **AVW** function call to extract the oblique image. Setting the interpolate variable to **AVW_TRUE**, causes the generation of a tri-linear interpolated image. A setting of **AVW_FALSE** causes the generation of a nearest neighbor interpolated image. The quality of the tri-linear image will be much better than the nearest neighbor image, but at the expense of being many times slower.

The example code in Figure 4-3 shows how **AVW_GetOblique()** is used to extract an oblique slice from a volume. If successful, an **AVW_Image** is returned. After processing, it's common for the image to be put either back into the original volume or to another volume with **AVW_PutOblique()**. Notice that the interpolate option is not available for **AVW_PutOblique()**, the nearest neighbor method is always used.

FIGURE 4-3 **Oblique Plane Example**

```
AVW_Image *image;
AVW_Volume *volume;
AVW_Matrix matrix;
  .
  .
  .
AVW_SetIdentityMatrix(matrix);
AVW_RotateMatrix(matrix, 45., 0., 0.);    /* 45 degree pitch */
AVW_RotateMatrix(matrix, 45., 90., 0.);   /* 90 degree roll  */

image = AVW_GetOblique(volume, matrix, AVW_TRUE, NULL);
if(!image) handle_error();
  .
  .
  .
if(!AVW_PutOblique(image, volume, matrix)) handle_error();
```

4.1.3 Curved Planes

Its possible that the structure of interest does not lie in a flat plane, but within a curved plane. The method **AVW** uses to describe a curved plane, is to start with a 2D orthogonal image through the structure of interest. A 2D trace is defined on top of this image. The curved plane is generated using the orientation and trace information. At each point along the trace, a row of pixels perpendicular to the orthogonal image, is used to generate each line of the curved image.

In Figure 4-4, an example of a call to **`AVW_GetCurved()`** is shown. The interactive tracing process, which most likely consists of a call to **`AVW_CreatePointList2()`** and many **`AVW_AddPoint2()`** calls inside an event loop, has not been included in this example.

FIGURE 4-4 **Curved Plane Example**

```
AVW_Image *image;
AVW_Volume *volume;
int orient = AVW_SAGITTAL;
AVW_Point2 point;
AVW_PointList2 *trace;
  .
  .
  .
image = AVW_GetCurved(volume, orient, trace);
if(!image) handle_error();
```

4.2 Basic Manipulations

Many processing applications require copying of an image, setting to a specific value, flipping an image, or shifting in a specific direction. **`AVW`** provides simple routine to accomplish all these tasks. These routines exists for volume manipulations also, but only the image routines will be discussed here.

4.2.1 Copying

The simple task of copying an image is done with **`AVW_CopyImage()`**. Provide the **`AVW_Image`** to be copied and this function returns an exact copy in a **`AVW_Image`**.

4.2.2 Setting to a Value

Many application will need to clear or erase an image. This is done by setting the entire image to a specific value. The value usually passed to **`AVW_SetImage()`** is zero, but can be anything. If the image is an **`AVW_COLOR`** image, the Red, Green, and Blue values need to be packed into one value before calling **`AVW_SetImage()`** (See Figure 4-5).

FIGURE 4-5 **Erasing RGB Image Example**

```
AVW_Image *image;

image = AVW_CreateImage(NULL, 100, 100, AVW_COLOR);
AVW_SetImage(image, AVW_MAKERGB(255, 255, 0));  /* Set to Yellow */
  .
  .
  .
```

4.2.3 Flipping

To flip an image horizontally, specify **`AVW_FLIPX`** for the axes parameter of **`AVW_FlipImage()`**. **`AVW_FLIPY`** flips vertically. OR **`(AVW_FLIPX | AVW_FLIPY)`** them together to flip on both axes at the same time.

4.2.4 Shifting

At times it may be necessary to shift an image horizontally or vertically in order to align structures from another image, or to correct for a wrap around problem during acquisition. The **`AVW_ShiftImage()`** function allows the specification of either a positive or negative vertical and/or horizontal shift. A wrap option indicates whether data shifted off one side wraps back onto the other, or is just lost.

4.3 Conversion

It is often necessary to convert images or volumes. **`AVW`** provides functions for changing data types, thresholding, intensity scaling, dithering, converting color images to gray scale, inverting, setting to a specific value, modifying using a table and converting an image to a volume.

NOTE: The following discussion apply to both images and volumes unless otherwise specified.

4.3.1 Changing Data Types

If a function, program, or data format does not support the **`DataType`** of an image, it can be converted to a **`DataType`** which is supported. The **`AVW_ConvertImage()`** function converts from any supported **`DataType`** to any specified **`DataType`**.

FIGURE 4-6 **DataTypes from AVW.h**

```
#define AVW_UNSIGNED_CHAR    (1<<0)
#define AVW_SIGNED_CHAR      (1<<1)
#define AVW_UNSIGNED_SHORT   (1<<2)
#define AVW_SIGNED_SHORT     (1<<3)
#define AVW_UNSIGNED_INT     (1<<4)
#define AVW_SIGNED_INT       (1<<5)
#define AVW_FLOAT            (1<<6)
#define AVW_COMPLEX          (1<<7)
#define AVW_COLOR            (1<<8)
```

If converting to a data type which holds less information, or information in a different range, the data is clipped according to the maximum and minimum values possible for the specified data type. No intensity scaling is performed during the conversion process. The rules of standard C type casting are followed.

If the input has a data type of **`AVW_COLOR`**, and the output data type is not **`AVW_COLOR`**, then a dithering process occurs during conversion.

If the input data has a colormap, it will be copied to the converted data. If the data type to be convert to is **`AVW_COLOR`**, the colormap is used in the conversion. Grayscale values converted to **`AVW_COLOR`**, result in the gray scale value being written to each channel of the **`AVW_COLOR`** image. Figure 4-7 shows a call to **`AVW_ConvertImage()`**, the call is the same, no matter if the input image has a colormap or not.

FIGURE 4-7 **Convert to RGB**

```
    AVW_Image *image;
      .
      .
      .
    image = AVW_ConvertImage(image, AVW_COLOR, image);
```

4.3.2 Thresholding

Intensity thresholding is a common segmentation technique. By specifying a maximum and minimum range of gray levels to be included in an object, that object can often easily be separated from other objects. This can be further enhanced by using seeded region growing. (See the Segmentation chapter for more detailed information.)

The example code in Figure 4-8 shows how **`AVW_ThresholdImage()`** can be used to change all non-zero pixels to 1's.

FIGURE 4-8 **Thresholding Example**

```
AVW_Image *image;
 .
 .
 .
image = AVW_ThresholdImage(image, 255., 1., image);
```

4.3.3 Intensity Scaling

Many display devices have a limit of 256 unique grayscale shades, and often all of those colors may not be available because of color usage on other parts of the screen. Most image modalities provide ranges of data outside this range, so conversion using intensity scaling for display is often necessary.

Assuming there are 128 colormap location available to display the grayscale image. These colormap locations start at color cell 100. The loaded volume is of data type unsigned short with an unknown maximum and minimum between 32767 and -32768.

Figure 4-9, illustrates how the **`AVW_FindImageMaxMin()`** function is used to find the volumes maximum and minimum values. These values are then used to scale each image before display. All values greater than the **`input_max`** are set to the **`output_max`**, all values less than the **`input_min`** are set to the **`output_min`**, and all values between the input maximum and minimum are linearly interpolated to be between the output maximum and minimums.

FIGURE 4-9 **Intensity Scaling Example**

```
AVW_Volume *volume;
```

```
    AVW_Image *image, *scaled_image=NULL;
    double input_max, input_min;
    double output_max = 227.;                /* 128 gray levels for */
    double output_min = 100;   /* display, starting at color cell 100 */
      .
      .
      .
    AVW_FindImageMaxMin(volume, &input_max, &input_min);
      .
      .
      .
    scaled_image = AVW_IntensityScaleImage(image, input_max, input_min,
                                           output_max, output_min,
                                           AVW_UNSIGNED_CHAR, image);
      .
      .
      .
```

4.3.4 Dithering

Two methods exist for handling color images within **AVW**. Colored images may be **AVW_UNSIGNED_CHAR** with an **AVW_Colormap**, or 24-bit **AVW_COLOR**, which consists of separate 8-bit images for the red, green, and blue channels. 24-bit color is the easiest and most powerful method of manipulating color images, but it's common for older or cheaper display devices to only have the capability of displaying the 8-bit with colormap.

Conversion from 24-bit color to 8-bit color often requires a technique which finds the closest color from a pre-defined colormap. This colormap can be created to either match the input image as close as possible, or it can be created to match any 24-bit image. The first method results in a better looking conversion for a single image, but would require a new colormap for each image converted. **AVW** uses the second method of creating a generic colormap that can be displayed with all the dithered images, but may not necessarily be the best for any particular image.

The size of the generic colormap is user selectable, and **AVW_DitherImage()** will generate the same colormap each time the same size is specified.

AVW uses a modified Floyd Steinberg dithering algorithm. When finding the "closest color", the difference from the actual color used is defused to the neighboring pixels.

NOTE: **`AVW_DitherVolume()`** does 2D dithering for each transverse slice in a volume.

4.3.5 Conversion to Gray Scale

Some displays, printers, and other devices only support gray scale. Color images sometimes need to be converted to gray scale. The **`AVW_MakeGrayImage()`** routine uses the following formula to convert to gray scale.

```
out_color = (red * .30) + (green * .59) + (blue * .11);
```

4.3.6 Inverting

Inversion of an image can done for a variety of reasons, displaying the negative and making masks are a few. The **`AVW_InvertImage()`** function takes a user supplied maximum and minimum value and inverts the pixels in the image using the following formula.

```
out_pixel = ((maximum - minimum) - (in_pixel - minimum)) + minimum;
```

4.3.7 Applying Tables

When requesting a series of colors from a windowing system, because of color sharing capability, the color cells returned are not always consecutive. In fact what one usually ends up with is a table of indexes. The **`AVW_TableImage()`** was created to applyi such a table to an image.

```
out_pixel = table[in_pixel];
```

Tables might also be useful as faster thresholding, scaling, or inverting methods.

4.3.8 Image to Volume

At times it may be desirable to treat a single image as a volume. The **`AVW_MakeVolumeFromImage()`** function accomplishes this.

4.4 Padding, Subregioning, and Resizing

`AVW` provides three ways to change the size of an image: padding, subregioning, and resizing. The following discussion deals with the image functions but corresponding volume functions are available as well.

4.4.1 Padding

Padding is done for a variety of reasons. One might be to increase the image size without changing the pixel dimensions. Another might be to allow functions to execute faster by avoiding special handling of edge pixels. A third reason, might be to add space to an image to insert another image, thus creating a side by side sequence. Use the **`AVW_PadImage()`** function specifying the desired width, height, and location.

4.4.2 Subregioning

Often image data will contain more structures than just the structure of interest. Subregioning removes the information outside a user specified region. Specify the region to extract in a **`AVW_Rect2`** and call **`AVW_GetSubImage()`**. See Figure 4-10.

The opposite of **`AVW_GetSubImage()`** is the **`AVW_PutSubImage()`** function. It can be used to insert one image into another.

FIGURE 4-10 **Extract the Center of an Image**

```
AVW_Image *image, *center_image;
AVW_Rect2 region;
int new_width, new_height;
  .
  .
  .
new_width = image->Width / 2;
new_height = image->Height / 2;

region.PointA.X = (image->Width / 2) - new_width / 2;
region.PointA.Y = (image->Height / 2) - new_height / 2;
region.PointB.X = region.PointA.X + new_width - 1;
region.PointB.Y = region.PointA.Y + new_height - 1;

center_image = AVW_GetSubImage(image, &region, NULL);
```

4.4.3 Resizing

Motivations for resizing images and volumes include: making the volume isotropic, reducing the memory usage, increasing the size to make things easier to see, or decreasing the size to allow an image to fit on the screen.

The **`AVW_ResizeImage()`** function provides an interpolate parameter. When enabled, bi-linear interpolation is performed to produce high quality images. When disabled, very fast pixel replicated or sub-sampled images are produced.

FIGURE 4-11 Magnify an Image 4 Times Using Replication

```
AVW_Image *image;
  .
  .
  .
image = AVW_ResizeImage(image, image->Width * 4., image->Height * 4.,
                        AVW_FALSE, image);
```

Because of the huge memory requirements often associated with volume resize operations, a special, slice-by-slice, memory efficient version of the volume resize is included in **`AVW`**. **`AVW_ResizeVolumeSliceBySlice()`** can be used "Disk to Memory" or "Disk to Disk" to resize a volume without requiring that both volumes be in memory.

4.5 Image and Volume Mathematics

`AVW` provides routines to complete mathematical operations between two images, between an image and a constant and also to apply a variety of mathematical functions to an image.

The operations (Figure 4-12) available include basic math, boolean and logical, along with a few functions.

FIGURE 4-12 AVW Mathematical Operators

Basic	Boolean	Logical	Functions
AVW_OP_ADD	AVW_OP_LT	AVW_OP_AND	AVW_F_SQRT
AVW_OP_SUB	AVW_OP_GT	AVW_OP_OR	AVW_F_LOG
AVW_OP_MUL	AVW_OP_LE		AVW_F_COUNT
AVW_OP_DIV	AVW_OP_GE		AVW_F_XPOS

Basic	Boolean	Logical	Functions
	AVW_OP_EQ		AVW_F_YPOS
AVW_OP_MOD	AVW_OP_NE		AVW_F_MAX
			AVW_F_MIN
			AVW_F_AVG
			AVW_F_ABS

4.5.1 Basic

The addition **`(AVW_OP_ADD)`**, subtraction **`(AVW_OP_SUB)`**, and multiplication **`(AVW_OP_MUL)`** work as expected. The division **`(AVW_OP_DIV)`** will return a division by zero error for the entire image if any voxel in the divisor is zero. The modulus **`(AVW_OP_MOD)`** operators puts the remainder into each voxel after a divide.

4.5.2 Boolean

The boolean operators will always result in an image of all ones and zeros. A one will occur each place in the output_image where the results of the compare is true. A zero will occur where the result is false. The boolean operations are: less than **`(AVW_OP_LT)`**, greater than **`(AVW_OP_GT),`** less than or equal to **`(AVW_OP_LE)`**, greater than or equal to **`(AVW_OP_GE)`**, equal **`(AVW_OP_EQ)`** and not equal **`(AVW_OP_NE)`**.

4.5.3 Logical

The logical operators result in the conversion to integer and the ANDing **`(AVW_OP_AND)`** or ORing **`(AVW_OP_OR)`** of the bits.

4.5.4 Functions

The **`AVW_FunctionImage()`** is used to apply functions to an image. If an output image is passed, it will be used, no matter what **`DataType`** it has. If **`NULL`** is passed, the returned image will have the same **`DataType`** as the input image. To insure against overflow, use **`AVW_CreateImage()`** to create an output image of the necessary type.

The square root **`(AVW_F_SQRT)`** for each pixel in the input image, is placed into the corresponding pixel location in the output image.

The natural log **`(AVW_F_LOG)`** is computed for each pixel in the input image and the results placed into the corresponding pixel location in the output image.

The count **(AVW_F_COUNT)** function operator, causes the current count to be placed into each pixel of the output image. The count starts at 1 and increases each time the count operator is used. The input image intensities are ignored, only the image size information is used.

The X position **(AVW_F_XPOS)** and Y position **(AVW_F_YPOS)** function operators, cause the current pixel location to be placed into each pixel of the output image. The position number starts at 1 for the first row or column and increases by one for each pixel along the row or column. The input image intensities are ignored, only the image size information is used.

The maximum **(AVW_F_MAX)** function operator, causes the entire input image to be examined, and the largest intensity value found is placed into each pixel of the output image.

The minimum **(AVW_F_MIN)** function operator, causes the entire input image to be examined, and the smallest intensity value found is placed into each pixel of the output image.

The average **(AVW_F_AVG)** function operator, causes the entire input image to be examined, and the calculated average intensity value is placed into each pixel of the output image.

The absolute value **(AVW_F_ABS)** function operator, causes the absolute value of each pixel in the input image to be placed into each pixel of the output image.

4.5.5 Constants

The **AVW_ImageOpConstant()** and **AVW_ConstantOpImage()** allow for operations between a specified constant and an image. If an output image is passed, it will be used, no matter what **DataType** it has. If **NULL** is passed, the returned image will have the same **DataType** as the input image. To insure against overflow, use **AVW_CreateImage()** to create an output image of the necessary type.

4.5.6 Images

The **AVW_ImageOpImage()** is used for operations between two images. If an output image is passed, it will be used, no matter what **DataType** it has. If **NULL** is passed, the returned image will have the same **DataType** as the largest (in size value it can hold) input image. To insure against overflow, use **AVW_CreateImage()** to create an output image of the necessary type.

FIGURE 4-13 **Add and Average All Images in a 16-bit Volume**

```
AVW_Image *image = NULL, *t_image;
AVW_Volume *volume;
int slice;
      .
      .
      .
for(slice=0; slice<volume->Depth; ++slice)
{
    image = AVW_GetOrthogonal(volume, AVW_TRANSVERSE, slice, image);
    if(slice)
    {
        t_image = AVW_ImageOpImage(t_image, AVW_OP_ADD, image, t_image);
    }
    else
    {                   /* float is big enough to hold all the info */
        t_image = AVW_ConvertImage(image, AVW_FLOAT, NULL);
    }
}

image = AVW_ImageOpConstant(t_image, ‘/’, (float) volume->Depth, image);
```

4.6 Matrix Manipulations

Most **AVW** functions require 4x4 matrices.

FIGURE 4-14 **AVW_Matrix**

```
AVW_Matrix *matrix;
matrix = AVW_CreateMatrix(4,4);
     .
     .
AVW_DestroyMatrix(matrix);
```

`AVW` provides routines to create, copy, destroy, invert, mirror, multiply, rotate, scale and translate matrices. **`AVW`** also provides a routine to set a matrix to the identity matrix. Matrices are used to describe rendering view points, light vectors and oblique image orientations. There are also routines which can apply a matrix to each pixel/voxel in an image/volume.

4.6.1 Identity Matrix

`AVW_SetIdentityMatrix()` fills each cell in the matrix to the appropriate values to indicates no translation, rotation or scaling. Note: **`AVW_CreateMatrix()`** always returns an indentity matrix.

```
AVW_Matrix *matrix;
matrix = AVW_CreateMatrix(4,4);
.
.
.
AVW_SetIdentity(matrix);
```

results:

```
matrix =   1.0  0.0  0.0  0.0
           0.0  1.0  0.0  0.0
           0.0  0.0  1.0  0.0
           0.0  0.0  0.0  1.0
```

4.6.2 Matrix Manipulations

`AVW_CopyMatrix()` is used to copy all cells in the input matrix to an output matrix. **`AVW_MultiplyMatrix()`** is used to multiply matrices (order is important when multiplying matrices). **`AVW_InvertMatrix()`** inverts a matrix. **`AVW_MirrorMatrix()`** alters the matrix so that a mirror image can be created. **`AVW_RotateMatrix()`**, **`AVW_ScaleMatrix()`** and **`AVW_TranslateMatrix()`** rotate, scale and translate a matrix respectively. When calling **`AVW_RotateMatrix()`**, the X angle is applied first, then Y and finally Z. These rotation will effect each other, so it is suggested that only one axis be rotated per **`AVW_RotateMatrix()`** call. Combinations of rotates and multiplies of the matrices result in rotations which are relative to the screen or volume.

4.6.3 Applying a Matrix to an Image

The **`AVW_TransformImage()`** routine can be used to transform each pixel in an image to a new location in a user specified output image. An interpolate option is available. Choose from tri-linear interpolated or the quicker, but less quality nearest neighbor. (See Figure 4-14)

Matrices may be applied to volumes with **`AVW_TransformVolume()`** or **`AVW_TransformVolumeSliceBySlice()`**. **`AVW_TransformVolume()`** transforms the entire input volume, while the slice by slice version allows the selection of any image from the output volume.

Whenever a matrix is applied, the origin of the input and output image is considered to be the center pixel or voxel. To apply rotation around a point which is not the center of the volume, translate to that point before rotating.

FIGURE 4-15 **Rotate an Image 45 Degrees Counter Clockwise Around Its Center.**

```
AVW_Image *image;
AVW_Matrix *matrix;
matrix = AVW_CreateMatrix(4,4);
.
.
.
AVW_SetIdentityMatrix(matrix);
AVW_RotateMatrix(matrix, 0., 0., -45.);

image = AVW_TransformImage(image, matrix, AVW_TRUE, image);
```

CHAPTER 5

Process Functions

The Process Functions include **AVW** functions that perform histogram operations and spatial filtering. There are both volume and image functions available for most of the Process routines.

5.1 Histogram Operations

Image manipulations based on the statistical distribution of gray scale may be used to correct contrast and brightness inconsistencies between images. They are particularly useful for pre-processing serial-section microscope images, and other situations where there is poor control over image capture parameters. The basic operation is histogram matching, which remaps the gray scale of an image so as to force its histogram to match a reference histogram. Usually the reference histogram is that of an “ideal” or “best available” image of the same subject. Histogram flattening is a special case of histogram matching where the reference histogram has an even (i.e. flat) distribution. Histogram preservation is an adaptation of histogram flattening designed to preserve as much information as possible when transforming an image from 16-bit to 8-bit gray scale.

5.1.1 Histogram Matching

In the first code example, a digitized photograph is to be converted into a plan for a tiled mosaic. The image consists of 128 X 128 pixels, and there are precisely:

5,166	Black Tiles
4,123	Dark Grey Tiles
3,827	Light Grey Tiles
3,268	White Tiles

available to build the mosaic. The code segment will force the distribution of gray scale values in the photograph to match the available distribution of tiles, regardless of the distribution or number of grey values in the original image.

FIGURE 5-1 **AVW_Histogram Structure from AVW_Histogram.h**

```
typedef struct
    {
    double *Mem;
    double Max;
    double Min;
    double Step;
    unsigned int Bins;/* Number of bins in the histogram */
    } AVW_Histogram;
```

FIGURE 5-2 **Match Image to Given Histogram**

```
    AVW_Image *in_image,*out_image=NULL;
    AVW_Histogram *histo;
      .
      .
      .
    histo = AVW_CreateHistogram(NULL,3.0,0.0,1.0);
    histo->Mem[0] = 5166.0;
    histo->Mem[1] = 4123.0;
    histo->Mem[2] = 3827.0;
    histo->Mem[3] = 3268.0;
    out_image = AVW_MatchImageHistogram(in_image, histo, out_image);
```

In the second code example, the first section of a volume is used as an “ideal” image to which the remaining sections are matched.

FIGURE 5-3 **Match Volume Histogram to Image**

```
AVW_Volume *volume;
AVW_Image *in_image=NULL, *out_image=NULL;
AVW_Histogram *histo=NULL;
  .
  .
  .
in_image = AVW_GetOrthogonal(volume, AVW_TRANSVERSE, 0, in_image);
histo = AVW_GetImageHistogram(in_image, NULL, NULL, histo);
for(i = 1 ; i < volume->Depth; i++)
{
     in_image = AVW_GetOrthogonal(volume, AVW_TRANSVERSE, i, in_image);
     out_image = AVW_MatchImageHistogram(in_image, histo, out_image);
     AVW_PutOrthogonal(out_image,volume, AVW_TRANSVERSE, i);
}
AVW_DestroyImage(in_image);
AVW_DestroyImage(out_image);
AVW_DestroyHistogram(histo);
```

5.1.2 Histogram Flattening

An image which has a flat histogram (i.e. roughly equal number of pixels at every grey level) shows maximal contrast over the entire coding range. The **`AVW_FlattenHistogramImage()`** function will force an image to be flat over a specified range and number of gray scales. The code example results in an image having a flat histogram with values in the range of -10 to 268. The input image must have a gray scale range greater than or equal to the range specified by **`max`** and **`min`**.

FIGURE 5-4 **Flatten Image Histogram**

```
AVW_Image *in_image, *out_image=NULL;
```

```
      .
      .
out_image = AVW_FlattenImageHistogram(in_image, 268.0, -10.0, out_image);
```

5.1.3 Histogram Preservation

The range of gray scale values in a 16-bit CT image is typically far smaller than the potential 16-bit range. In addition to this simple linear “coding range inefficiency”, the high degree of spatial and gray scale coherence in a meaningful image reveals a further “statistical coding inefficiency” in terms of the information content of the image.

For example, a 16-bit CT image of the chest may only contain gray scale values in a 12-bit range. Furthermore, since the image contains fine gray scale detail image information in the bright spinal bone region as well as in the dark lung field and the mid-grey soft tissue - well separated ranges of gray scale with very few pixels at values between the ranges - the statistical variability of the image (i.e. information content or *entropy* (reference 1)) may be no more than 8 bits.

Linear range rescaling merely propagates the coding inefficiencies. So if the entire 16-bit range of the image was rescaled to the 8-bit range, there would only be values in a 6-bit range, and the remaining information content would be 4 bits. If the 12-bit range was rescaled to the 8-bit range, the range coding inefficiency would be overcome, but the non-uniform statistical distribution of gray scale would still reduce scaled image information content to about 6.6 bits. If the same relative contrast *between* tissue ranges is maintained by linear scaling to a lower gray scale resolution, the fine detail *within* each range is lost.

A process of slope-limited histogram flattening (“histogram preservation”) will automatically seek the best possible monotonic mapping of a higher gray scale resolution image to a lower resolution. The resulting image will have a statistical information content very near the target coding range or the original image entropy, whichever is less. Many 16-bit CT and MR images may be coded to 8-bits with no loss of visual or spatial analysis accuracy. Note that the mapping is non-linear, and may complicate analysis of absolute image gray scale value.

FIGURE 5-5 **Preserve Volume Histogram**

```
AVW_Volume *in_vol, *out_vol=NULL;
      .
      .
out_vol = AVW_PreserveVolumeHistogram(in_vol, 268.0, -10.0, out_vol);
```

5.2 Spatial Filter Operations

A number of linear and non-linear spatial filters for both images and volumes are available in **AVW**. Spatial convolution forms the basis of the linear filters.

The convolution integral:

$$\sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty} Image(x, y)PSF(n-x, m-y) \qquad \text{(EQ 1)}$$

produces the output due to the input *Image* of a linear image process which responds to a point source of light with the image *PSF*. Because the convolution integral is itself a linear system, there in no loss of information in convolution with any PSF, and all convolutions are reversible.

Convolving with a small PSF may be efficiently performed in the spatial domain, but convolution with a PSF of unknown size or that is a significant fraction of the image size is more efficiently performed in the Fourier domain. Deconvolution is generally performed only in the Fourier domain.

The non linear spatial filters may be thought of as similar to convolution in that a small moving region of the image is considered for calculation of each output pixel value. Since the calculation or decision is non-linear, the operations generally result in a loss of information and are not reversible.

5.2.1 Spatial Convolution

In the first code example, an eye chart image is convolved with a small PSF representing the optical response of a human cornea. The result represents the quality of the retinal image.

FIGURE 5-6 Convolve Image Example 1

```
AVW_Image *Image,*Result,*PSF;
AVW_ImageFile *file;
      .
      .
    file = AVW_OpenImageFile(“eyechart”, ”r”);
    Image = AVW_ReadImageFile(file, NULL);
    AVW_CloseImageFile(file);
    file = AVW_OpenImageFile(“PSF”, ”r”);
    PSF = AVW_ReadImageFile(file, NULL)
    AVW_CloseImageFile(file);
    Result = AVW_ConvolveImage(Image, PSF, NULL);
```

The 3D version of this function may be used to convolve a 3D Image with a 3D PSF, but may also be used to convolve each section of a volume with a 2D PSF. The next two code examples illustrate this property and have identical results.

FIGURE 5-7 Convolve Image Example 2

```
AVW_Image *PSF,*section=NULL,*result=NULL;
AVW_Volume *Volume,*Result;
AVW_ImageFile *file;
      .
      .
file = AVW_OpenImageFile(“volume”, ”r”);
Volume = AVW_ReadVolume(file, 0, NULL);
AVW_CloseImageFile(file);
file = AVW_OpenImageFile(“PSF”, ”r”);
PSF = AVW_ReadImageFile(file, NULL)
AVW_CloseImageFile(file);

Result = AVW_CreateVolume(NULL, Volume->Width, Volume->Height,
                                 Volume->Depth, Volume->DataType);
for(i = 0 ; i < Volume->Depth ; i++)
{
      section = AVW_GetOrthogonal(Volume, AVW_TRANSVERSE, i,section);
      result = AVW_ConvolveImage(section, PSF, result);
      AVW_PutOrthogonal(result, Result, AVW_TRANSVERSE, i);
}
```

FIGURE 5-8 Convolve Volume Example

```
    AVW_Volume *Volume, *Result, *PSF;
    AVW_ImageFile *file;
```

```
      .
      .
    file = AVW_OpenImageFile(“volume”, ”r”);
    Volume = AVW_ReadVolume(file, 0, NULL);
    AVW_CloseImageFile(file);
    file = AVW_OpenImageFile(“PSF”, ”r”);
    PSF = AVW_ReadVolume(file, NULL)
    AVW_CloseImageFile(file);

    Result = AVW_ConvolveVolume(Volume, PSF, NULL);
```

5.2.2 Simple Convolution Filters

The **`AVW_LowpassFilterVolume()`** is performed by spatial convolution with a uniform rectilinear region.

If the voxel value (gray scale), at coordinate (x,y,z) is denoted as v(x,y,z), and the filter window dimensions are (fx,fy,fz), so that:

$$fx = 1 + 2 \cdot dx$$

$$fy = 1 + 2 \cdot dy$$

$$fz = 1 + 2 \cdot dz \quad ,$$

Then the **`AVW_LowpassFilterVolume()`** output at coordinate (x,y,z) is:

$$v_l(x, y, z) = \frac{\sum_{k=-dz}^{dz} \sum_{j=-dy}^{dy} \sum_{i=-dx}^{dx} v(x+i, y+j, z+k)}{fx \cdot fy \cdot fz} \qquad \text{(EQ 2)}$$

`AVW_LowpassFilterVolume()` therefore replaces a voxel with the average of voxel values in a rectilinear region centered about the voxel. The amount of blurring in a particular direction is proportional to extent of the region in that dimension. The code example blurs the volume image significantly in x, less so in y, and does not blur at all in the z direction.

FIGURE 5-9 5 X 3 Lowpass Filter

```
AVW_Volume *Volume, *Result;
int extents[3];
  .
  .
extents[0] = 5;
extents[1] = 3;
extents[2] = 1;
Result = AVW_LowpassFilterVolume(Volume, extents, NULL);
```

`AVW_UnsharpFilterVolume()` effects high-pass filtering (edge filtering) by subtracting the lowpass filtered volume from the original:

$$v_u(x, y, z) = v(x, y, z) - v_l(x, y, z) \qquad \text{(EQ 3)}$$

`AVW_UnsharpFilterEnhanceVolume()` adds the edge-filtered volume back to the original volume to enhance edge information:

$$v_{ue} = v(x, y, z) + v_u(x, y, z) \qquad \text{(EQ 4)}$$

Using the same extents used in the **`AVW_LowpassFilterVolume()`** code example for an **`AVW_UnsharpFilterEnhanceVolume()`** operation will enhance edges significantly in x, less so in y, and does not edge-enhance at all in the z direction.

FIGURE 5-10 5X3 Enhanced Unsharp Filter

```
AVW_Volume *Volume, *Result;
int extents[3];
  .
  .
extents[0] = 5;
extents[1] = 3;
extents[2] = 1;
Result = AVW_UnsharpFilterEnhanceVolume(Volume, extents, NULL);
```

5.2.3 Other Edge Filters

There are two other edge-filters available in AVW. **`AVW_SobelFilterVolume()`**is defined by separate convolution with three different edge-detecting kernels for each of the x, y, and z dimensions:

$$w_x(i, j, k) = \frac{i}{(i^2 + j^2 + k^2)}$$

$$w_y(i, j, k) = \frac{j}{(i^2 + j^2 + k^2)}$$

$$w_z(i, j, k) = \frac{k}{(i^2 + j^2 + k^2)}$$

with the total filter weight is defined as:

$$weight = \sum_{k=-dz}^{dz} \sum_{j=-dy}^{dy} \sum_{i=-dx}^{dx} |w_x(i, j, k)| + |w_y(i, j, k)| + |w_z(i, j, k)| \quad \text{(EQ 5)}$$

The *x*-direction Sobel filter is then:

$$S_x(x, y, z) = \sum_{k=-dz}^{dz} \sum_{j=-dy}^{dy} \sum_{i=-dx}^{dx} w_x(i, j, k) \cdot v(x + i, y + j, z + k) \quad \text{(EQ 6)}$$

and the *y* and *z* weights are similarly computed. The final output of the *Sobel* filter is:

$$v_s(x, y, z) = \sqrt{\frac{S_x(x, y, z)^2 + S_y(x, y, z)^2 + S_z(x, y, z)^2}{weight}} \quad \text{(EQ 7)}$$

A**`VW_SobelFilterVolume()`** may be used with variably-sized rectangular extents as the previous filters, and **`AVW_SobelFilterEnhanceVolume()`** performs edge-enhancement in a manner similar to **`AVW_UnsharpFilterEnhanceVolume()`**:

$$v_{se}(x, y, z) = v(x, y, z) + v_s(x, y, z) \quad \text{(EQ 8)}$$

`AVW_OrthoGradFilterVolume()` is a fast approximation of the magnitude of the local gray scale gradient, calculated as the maximum absolute difference between immediate orthogonal neighbors The orthogonal gradients are defined as:

$$G_x(x, y, z) = |v(x+1, y, z) - v(x-1, y, z)| \quad \text{(EQ 9)}$$

$$G_y(x, y, z) = |v(x, y+1, z) - v(x, y-1, z)| \quad \text{(EQ 10)}$$

$$G_z(x, y, z) = |v(x, y, z+1) - v(x, y, z-1)| \quad \text{(EQ 11)}$$

The maximum orthogonal gradient filter is defined as:

$$v_g(x, y, z) = max(G_x(x, y, z), G_y(x, y, z), G_z(x, y, z)) \quad \text{(EQ 12)}$$

5.2.4 Non-Linear Filters

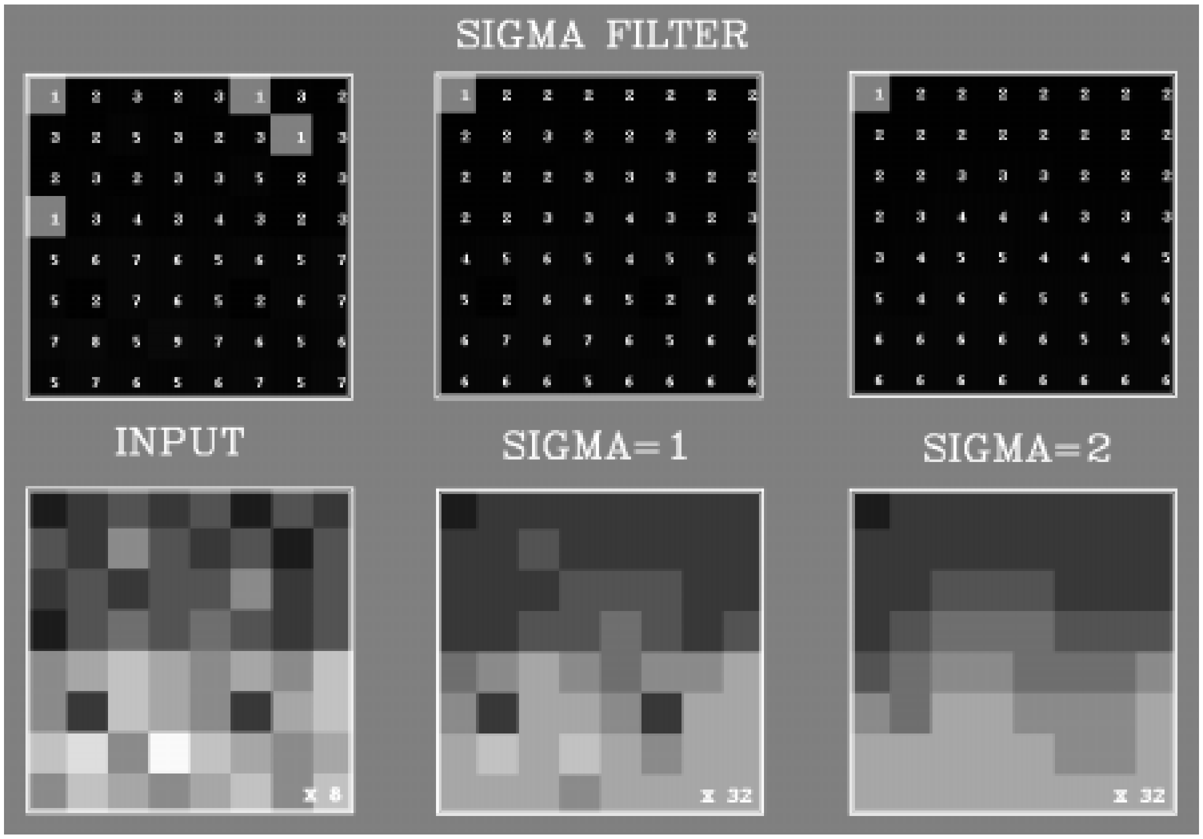
A number of useful image filters may be described as single-pass non-linear neighborhood operators. Like convolution-based operators the output at a pixel coordinate is dependent upon a variable-sized neighborhood about that pixel, but unlike convolution, the neighborhood operation is nonlinear.

`AVW_SigmaFilterImage()` has two variable parameters. **`Extents`** defines the effective neighborhood, as in the previous examples. **`Sigma`** represents a “similarity criterion”. For each pixel, all neighboring pixels which differ from the input pixel by less than 2*Sigma are averaged to produce the output pixel. A priori knowledge of different tissue types represented by different ranges of image pixel values can be used to remove spot noise and homogenize image regions of uniform underlying tissue type.

FIGURE 5-11 3 X 3 Sigma Filter

```
AVW_Image *Image, *Result;
int extents[2];
  .
  .
extents[0] = 3;
extents[1] = 3;
Result = AVW_SigmaFilterImage(Image, extents, 1, NULL);
Result = AVW_SigmaFilterImage(Image, extents, 2, Result);
```

FIGURE 5-12 **AVW_SigmaFilterImage Results**



Note that the upgoing and downgoing spikes have been preserved because they are greater than 2*sigma different than their background. Larger neighborhoods tend to increase the amount of smoothing, but will not show appreciable effects on so small an image.

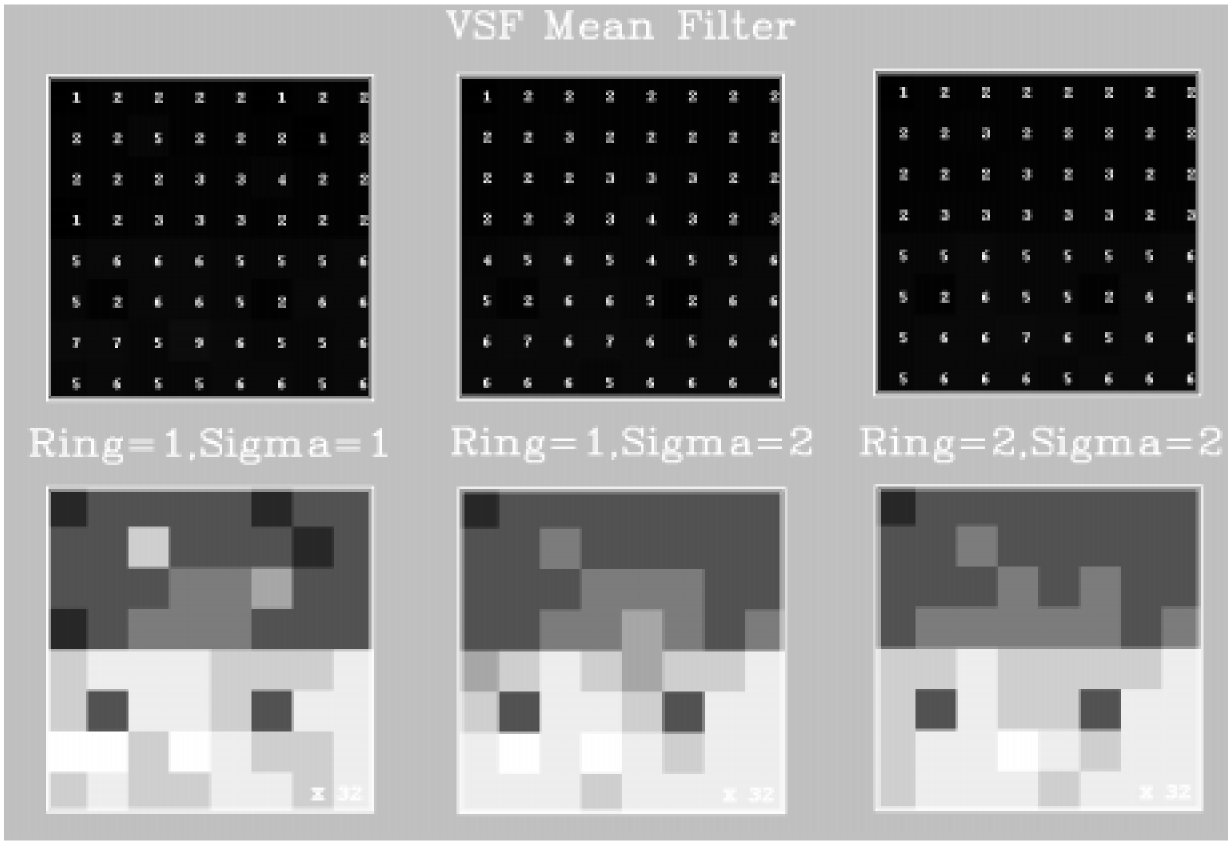
When the `sigma` value is increased to 2, the single-pixel spikes are removed

`AVW_VSFMeanfFilter()` is an alternative implementation of the **`sigma`** filter. The program uses +- sigma (rather than +- 2sigma) as the similarity constraint, providing closer control on the averaging function, and specifies filter size by a distance parameter, “ring”, which specifies the maximum distance of a neighbor from the central pixel. Therefore a Ring=1,Sigma=2 VSF mean filter should be identical to the Sigma=1 3X3 Sigma filter.

FIGURE 5-13 **VSF Mean Filter**

```
AVW_Image *Image,*Result;
  .
  .
Result = AVW_VSFMeanfFilterImage(Image, 1, 1, NULL);
Result = AVW_VSFMeanfFilterImage(Image, 1, 2, Result);
Result = AVW_VSFMeanfFilterImage(Image, 2, 2, Result);
```

FIGURE 5-14 **VSF Mean Filter Results**

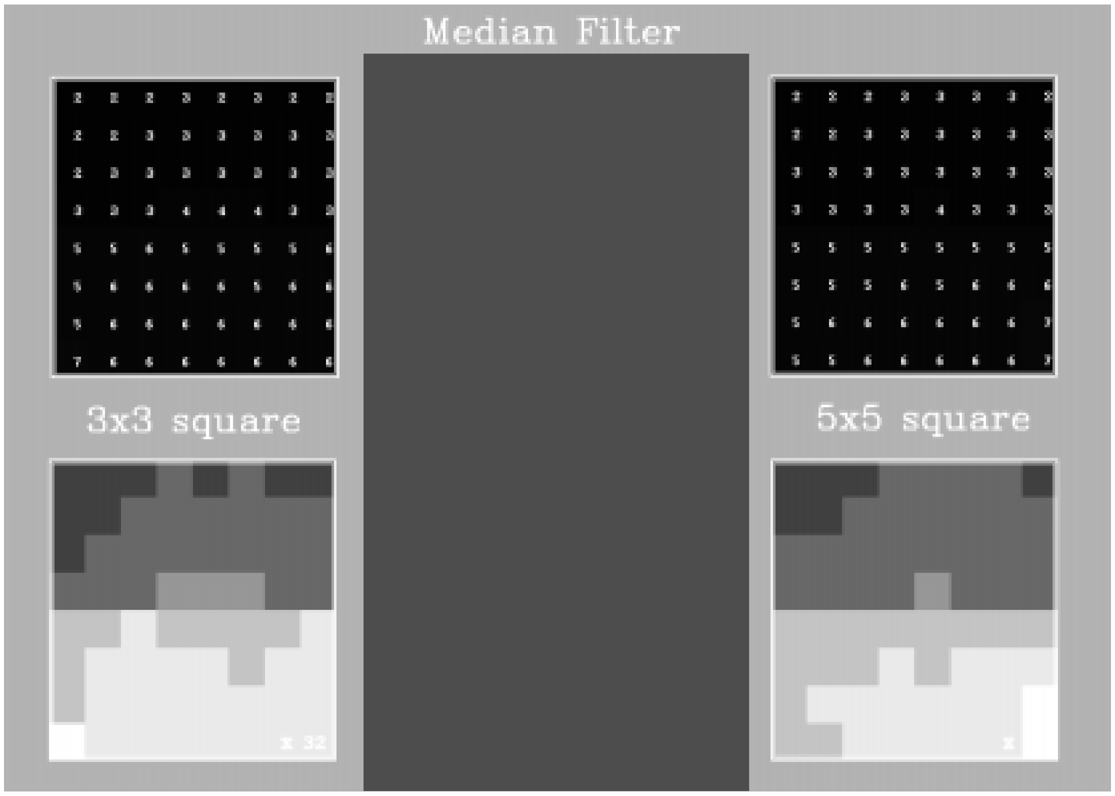


Another useful non-linear neighborhood operator is the rank function. The rank function simply orders the pixels in the neighborhood from smallest to largest and then uses the pixel at a preselected rank as the output. The most common type of rank filter returns the median value from the neighborhood. This filter also eliminates single-pixel spikes, while preserving edges:

FIGURE 5-15 **Rank Filter**

```
AVW_Image *Image, *Result;
int extents[2];
  .
  .
extents[0] = 3;
extents[1] = 3;
Result = AVW_RankFilterImage(Image, extents, 0, NULL);
extents[0] = 5;
extents[1] = 5;
Result = AVW_RankFilterImage(Image, extents, 13, Result);
```

FIGURE 5-16 **Rank Filter Results**



CHAPTER 6

Fourier Transforms and Deconvolution

There are a variety of **`AVW`** routines which perform 2D and 3D forward and inverse Fast Fourier Transforms (FFTs), convolutions, and deconvolutions on image data. When a forward FFT is performed on an image, the result is the frequency spectrum of the image. This spectrum can be filtered (multiplied by a filter function) to enhance some frequencies and suppress others. Such filters can smooth, sharpen or perform more complicated operations on the image. Specifically, convolution or deconvolution of the image by a second function can be performed by multiplying or dividing the spectrum by a filter which is the Fourier transform of the second function and then performing an inverse FFT to obtain the convolved or deconvolved image. Advanced deconvolution techniques (nearest neighbor slice-by-slice deconvolution, constrained iterative deconvolution and maximum likelihood deconvolution) commonly used in optical and confocal microscopy are also available.

6.1 Fourier Transforms

Forward and inverse Fourier transforms of an image or volume can be carried out with the **`AVW_FFT2D()`** and **`AVW_FFT3D()`** routines. For a forward FFT, image dimensions are automatically padded to the next larger power of 2 before the transform is applied. If these (possibly padded) dimensions are, say, X2 and Y2, the returned image will be an image of data type **`AVW_COMPLEX`** with dimensions X2/2 + 1, Y2. For an inverse FFT, the input image must be of data type **`AVW_COMPLEX`** and of dimensions X2/2 + 1, Y2 where X2 and Y2 are powers of 2. The returned image will be of data type **`AVW_FLOAT`** and of dimensions X2, Y2.

6.2 Frequency Space Convolution

The convolution of two images (or of an image and a filter) can be obtained by performing forward FFTs to obtain the spectra of the images, multiplying the spectra together with a call to **`AVW_ImageOpImage()`**, and taking the inverse transform of the result. The sizes of the two images (or image and filter) must be compatible.

FIGURE 6-1 **FFT and Convolution Example**

```
AVW_Image *image1, *image2, *image_out;
AVW_Image *sp1, *sp2, *sp_out;
     .
     .
     .
sp1 = AVW_FFT2D(image1, AVW_FORWARD, NULL);
sp2 = AVW_FFT2D(image2, AVW_FORWARD, NULL);
sp_out = AVW_ImageOpImage(sp1, AVW_OP_MULT, sp2, NULL);
image_out = AVW_FFT2D(sp_out, AVW_BACKWARD, NULL);
```

6.3 Filter Creation

Some classic filters (Gaussian and Butterworth filters) can be created with the **AVW** routines **AVW_CreateButterworthCoeffs()** and **AVW_CreateGaussianCoeffs()**. These routines actually create a 1D array of coefficients, which are swept around to fill in a 2D or 3D array with the **AVW_CreateCircularMTF()** and **AVW_CreateSphericalMTF()** routines. The output of these routines are transfer functions; that is, they are already in the spectral domain, so they can be directly multiplied by the spectrum of the image and no forward FFT needs to be done to them. A circularly or spherically symmetric filter with arbitrary coefficients can be created by passing a list of numbers into **AVW_CreateCoeffs()** and proceeding as above. It is also possible to create a filter which corresponds to the modulation transfer function of an optical microscope, with the **AVW_CreateStoksethMTF()** routine.

FIGURE 6-2 Gaussian Filter Example

```
AVW_Image *image, *image_out;              /* assume image is 512 x 512 */
AVW_Image *spectrum, *sp_filt, *sp_out;
AVW_FilterCoeffs gaus_coeffs;
double dev = 8.0;
int num_samples = 257;                          /* note this is 512/2+1 */
     .
     .
     .
gaus_coeffs = AVW_CreateGaussianCoeffs(dev, num_samples, NULL);
sp_filt = AVW_CreateCircularMTF(gaus_coeffs, NULL);
spectrum = AVW_FFT2D(image, AVW_FORWARD, NULL);
```

```
sp_out = AVW_ImageOpImage(sp_filt, AVW_OP_MULT, spectrum, NULL);
image_out = AVW_FFT2D(sp_out, AVW_BACKWARD, NULL);
```

6.4 Deconvolution

Let us assume we have an image from some sort of imaging system (e.g., an optical or confocal microscope) and one is interested in deconvolving or “deblurring” the image to observe greater detail. The image can be 2D or 3D; with optical or confocal microscopy, a 3D volume would be built up from a set of 2D slices taken while advancing the focal plane in the z direction. In either case, if one can measure or estimate the “point spread function” for the system - i.e., the output from a single infinitely small point source - and if this point spread function does in fact accurately characterize the system’s response (across the entire field of view, at the conditions under which the image was taken, etc.), then deconvolution can be readily carried out.

Ideally, a forward FFT would be performed on the image and on the point spread function (this latter spectrum is called the “transfer function” of the imaging system), divide the former by the latter to perform the deconvolution, and do an inverse FFT on the result to get the deconvolved image. Unfortunately, this simple approach is very sensitive to noise, and does not really work in a practical sense. This can be seen when one considers that dividing by the filter involves dividing by some very small numbers (usually very many, at the higher frequencies), which is equivalent to multiplication by very large numbers. Thus, any noise or uncertainty gets greatly amplified, and the problem is ill-conditioned.

Many algorithms exist to provide practical solutions to this problem, with a wide variety of trade-offs between accuracy and calculation time. The approaches described below are available as **`AVW`** routines. The latter two (constrained iterative deconvolution and nearest-neighbor deconvolution) are implemented in the manner described in reference [1], which should be considered required reading.

Deconvolution of images (especially 3D deconvolution) can be very intensive in terms of both computer memory and processing power. Also, there are several things of which one must be careful. Because of the cyclical nature of the FFT, the image is treated as if it wraps around and opposite edges are adjacent. If there are intensity mismatches between such opposite edges in the image being deconvolved, these will produce undesirable “wraparound” effects in the result. The image should be windowed, or scaled to zero smoothly at all the edges, to minimize these effects. If the psf image is not centered, then any deconvolved image will be translated relative to the original image, in the opposite direction to and by the amount off-center in each dimension. In particular, the iterative deconvolution algorithms will be affected by this and will yield improper results.

6.4.1 Simple thresholding

The simplest scheme is to limit how small the filter values can be (or how large the factor “1/filter value” by which we are multiplying the spectrum can be). This can be accomplished with the **`AVW_DeconvDivideImage()`** and **`AVW_DeconvDivideVolume()`** routines. Briefly, these routines divide the image spectrum by the filter spectrum, but as the division is carried out, the magnitude of each filter value is checked and, if smaller than the supplied **`fmin`** parameter, altered appropriately. This approach is only effective in nearly noise-free situations, and is presented here for comparison purposes.

6.4.2 Wiener Deconvolution

A commonly used algorithm is Wiener deconvolution, which corresponds to multiplying the spectrum buffer by $F^* / (|F|^2 + \alpha^2)$, where F represents the transfer function and α is a parameter set by the user based on the relative noise level of the data. The value of α should be high (say 0.5) for very noisy data and low (say 0.01) for very clean data. A true Wiener deconvolution provides near-optimal linear filter noise suppression and is far more complicated than this. The above is only an approximation, but a common one which is useful in practice, and is implemented here in both 2D and 3D in **`AVW_DeconvWienerImage()`** and **`AVW_DeconvWienerVolume()`**.

FIGURE 6-3 **Wiener Deconvolution**

```
AVW_Image *image;                             /* appropriately windowed */
AVW_Image *psf_image;                         /* appropriately centered */
AVW_Image *spectrum, *transfer_func, *result_sp, *result;
double alpha;
     .
     .
     .
alpha = 0.2;
spectrum = AVW_FFT2D(image, AVW_FORWARD, NULL);
transfer_func = AVW_FFT2D(psf_image, AVW_FORWARD, NULL);
result_sp = AVW_DeconvWienerImage(image, transfer_func, alpha, NULL);
result = AVW_FFT2D(result_sp, AVW_BACKWARD, NULL);
```

6.4.3 Constrained Iterative Deconvolution

The Wiener filter is a near-optimal linear restoration method for recovering a signal in the presence of noise but, like all linear methods, necessarily suffers from "ringing" - negative ripples that build up around strong features, caused by the fundamentally band-limited nature of the methods. Nonlinear restoration methods that constrain the result to be positive can eliminate this problem, infer information that was present in higher frequencies, and provide higher resolution in the result. Perhaps the simplest such method, constrained iterative deconvolution, is implemented here, in both 2D and 3D, in the **`AVW_IterDeconvImage()`** and **`AVW_IterDeconvVolume()`** routines. We follow the treatment described in [1], section C.

The basic idea of the method is to make a guess at the restored image, convolve this guess with the point spread function, and note the differences between the result and the original observed image. These differences are used to update the guess, the updated guess is constrained to be positive, and then the process repeats. The method is stable and good convergence is usually reached in 5-10 iterations. Both the *Van Cittert* and *Gold* update methods mentioned in [1] are available; we have found the former to be more practical. As in [1], the initial guess can be either the observed image itself, or a Wiener-filtered version. Also as in [1], we have found that smoothing the guess after several iterations with a small Gaussian filter yields better results.

6.4.4 Nearest Neighbor Deconvolution

The technique of nearest neighbor deconvolution for deblurring optical section data in one focal plane by using data from the slices above and below is also implemented here, following the treatment of [1], section A. This is a 2D method, applied to successive slices one at a time, which tries to make corrections for blurring in the third dimension using information from (and transfer functions appropriate to) the adjacent slices. It is less accurate but faster than the techniques above, and approximates a 3D deconvolution while requiring less memory than a true 3D deconvolution.

The basic equation (adapted from [1]) is

$$I_j = [O_j - c(O_{j-1} + O_{j+1})S_1]\frac{S_o^*}{|S_o|^2 + \alpha^2}$$

where S_o is the in-focus transfer function, S_1 is the one slice out-of-focus transfer function, O_j is the spectrum of the current slice, O_{j-1} and O_{j+1} are the spectra of the slices above and below the current slice, I_j is the desired deblurred spectrum, α performs the same function as in Wiener deconvolution above, and ***c*** is a constant (0.45 is a recommended value) which weights the out-of-focus removal. The implementation is in the **`AVW_NearestNeighborDeconv()`** routine.

6.4.5 Sample programs

A few sample programs are included in **`$AVW/extras`**.

References

1. Agard, David A., Hiraoka, Yasushi, Shaw, Peter, and Sedat, John W., "Fluorescence Microscopy in Three Dimensions", 1989, in Methods in Cell Biology, Vol. 30, Chap. 13, Academic Press.

CHAPTER 7 Segmentation

Segmentation is the process of separating a meaningful object or feature from the rest of the data within an image or volume. The segmentation process is usually very user intensive and requires the development of an extensive user interface. **AVW** provides segmentation routines upon which these interfaces can be built. Segmentation can be achieved with **AVW** by thresholding, region growing and morphology. Multispectral classification can also be used to segment volumes if information from different imaging modalities is available (See Chapter 11).

7.1 Thresholding

Gray level thresholding is a useful segmentation technique if an object or feature is defined by a distinct range of gray level intensities. The thresholding operation sets all of the voxels greater than or equal to the threshold minimum and less than or equal to the threshold maximum to 1 and all other voxels to 0.
AVW_ThresholdVolume() and **AVW_ThresholdImage()** are the **AVW** routines which support thresholding.

FIGURE 7-1 **Threshold Example**

```
AVW_Volume *volume;
AVW_Volume *t_volume=NULL;
double t_max = 255.0;
double t_min = 53.0;
     .
     .
     .
t_volume = AVW_ThresholdVolume(volume, t_max, t_min, t_volume);
```

In some cases thresholding is all that is needed for segmentation. The thresholded image or volume can be used as a mask in the measurement routines or as an object for visualization. In other cases region growing or morphology techniques may need to be applied to thresholded images or volumes to fully isolate the object or feature.

7.2 Region Growing

Another useful segmentation technique that can be applied in conjunction with thresholding is that of region growing. In this technique objects are segmented or grouped by their connectivity. Starting with one pixel, neighbors are recursively examined to see if they are connected to the "seed" pixel, and thus belong to the same object. The neighbors to be examined can vary as follows. For images, there are four connected and eight connected neighbors.

FIGURE 7-2 Four and Eight Connected Neighbors

```
0 1 0        1 1 1
1 * 1        1 * 1
0 1 0        1 1 1
```

* represents the seed pixel.

For volumes, neighbors can be six or twenty six connected. Voxels on the image below and above the current image are also examined.

FIGURE 7-3 Six and Twenty Six Connected Neighbors

```
0 0 0       0 1 0       0 0 0
0 1 0       1 * 1       0 1 0
0 0 0       0 1 0       0 0 0


1 1 1       1 1 1       1 1 1
1 1 1       1 * 1       1 1 1
1 1 1       1 1 1       1 1 1
```

The following connectivity parameters are defined in **`AVW.h`** and should be used when calling the **`AVW`** region growing functions.

FIGURE 7-4 **Neighbor Definitions from AVW.h**

```
#define AVW_4_CONNECTED    0
#define AVW_8_CONNECTED    1

#define AVW_6_CONNECTED    0
#define AVW_26_CONNECTED   1
```

Region growing is supported in **`AVW`** by a number of routines. **`AVW_FindImageComponents()`** and **`AVW_FindVolumeComponents()`** take a thresholded image or volume and find all of the distinct connected components. The components can be labelled, i.e. each component given a unique value, and sorted according to their size. A minimum and maximum size can also be specified and components whose size is not within these bounds will be ignored.

`AVW_ConnectAndDeleteImage()` and **`AVW_ConnectAndDeleteVolume()`** take a gray scale image or volume, a list of seed points, a threshold maximum and minimum, and a “delete value” as input parameters. All pixels or voxels within the threshold range and connected to the seed points are set to the “delete value” in the returned image or volume. A count of all the deleted pixels or voxels is also computed and returned.

`AVW_ConnectAndKeepImage()` and **`AVW_ConnectAndKeepVolume()`** set the unconnected pixels or voxels to the “delete value”.

`AVW_GetThresholdedBoundary()` returns a list of points which represent the boundary of a thresholded, four connected region. The repetitive display of these points on top of the gray scale image while adjusting the threshold maximum and minimum can reproduce the “auto trace” functionality contained in **ANALYZE™™**.

7.3 Mathematical Morphology

Mathematical morphology techniques offer a shape based approach to segmentation. Images or volumes are processed with a structuring element, which is a small image or volume. The structuring element is translated so that it is centered on every point of the image or volume and based on the voxels in the structuring element, the operation being performed and the neighboring voxels in the volume the point can be set to 1 or 0. Most of the morphology routines work on binary valued **`AVW_UNSIGNED_CHAR`** data. Thus, a thresholding step is usually a prerequisite.

The first basic morphological operation is an erode. An erode is performed by translating the structuring element, which is made up of 1s and 0s, so that its center lies at every point of the data being processed. At each point, if the point in the data being processed is 1, and any 1 in the structuring element corresponds to a zero in the data, the point is set to 0. The erode process can be thought of as peeling a layer away from an object.

A related morpological operation is dilate. A dilate is performed by translating the structuring element in the same manner as for an erode. If the point in the data being processed is 0 and there is a 1 corresponding to a 1 in the structuring element the point is set to 1. Dilates can also be conditional, that is the point is only set to 1 if it meets the prior conditions and it is also a 1 in a separate conditional volume. If a volume was thresholded and eroded, a conditional dilate with the original thresholded volume, ensures that voxels not in the original threshold range are not added during the dilate.

An open is defined as an erode followed by a dilate and a close is defined as a dilate followed by an erode.

Mathematical morphology is supported in **AVW** by the following routines:

AVW_CreateStructuringImage() and **AVW_CreateStructuringVolume()** allow the creation of simple structuring elements. **AVW_CreateImage()**, **AVW_CreateVolume()**, **AVW_PutPixel()**, and **AVW_PutVoxel()** can also be used to create other structuring elements.

AVW_ErodeImage(), **AVW_ErodeVolume()**, **AVW_DilateImage()**, **AVW_DilateVolume()**, **AVW_ConditionalDilateImage()**, **AVW_ConditionalDilateVolume()**, **AVW_MorphOpenImage()**, **AVW_MorphCloseImage()** and **AVW_MorphCloseVolume()** perform the basic 2D and 3D morphology operations described above.

AVW_UltimateErosionImage() and **AVW_UltimateErosionVolume()** perform successive erosions on an image or volume until all structures are removed. Each unique connected component, either in the data originally or created through the erosion process, is labelled with a value equal to the number of erosions required to fully erode away the component.

AVW_NonMaxImage() and **AVW_NonMaxVolume()** set all voxels which are not a maximum in the local neighborhood defined by the structuring element to 0.

AVW_MorphMaxImage() and **AVW_MorphMinImage()** are gray scale versions of the dilate and erode routines. The voxels in the processed data are set to the maximum or minimum value corresponding to a set voxel in the structuring element.

7.4 Other Segmentation Routines

There are a variety of other routines that may be useful in a segmentation process. **`AVW_Thin2D()`** and **`AVW_Thin3D()`** can be used to thin structures. **`AVW_FillHolesImage()`** and **`AVW_FillHolesVolume()`** may be used to fill voids in segmented structures. **`AVW_FindImageEdges()`** and **`AVW_FindVolumeEdges()`** isolate the edge points of an object. **`AVW_LabelImageFromEdges()`** and **`AVW_LabelVolumeFromEdges()`** assigns unique labels to all of the connected components defined by edges.

7.5 Sample Code

The following code shows an example of a segmentation process. Complete examples programs of segmentation are included in **`$AVW/extras`**.

FIGURE 7-5 **Segmentation Example**

```
AVW_Volume *volume1=NULL, *volume2=NULL, *element, *volume3;
double t_max = 255.0, t_min = 53.0;
     .                                         /* Read volume from Disk */
     .
     .                                              /* Threshold Volume */
volume2 = AVW_ThresholdVolume(volume1, t_max, t_min, volume2);

                     /* Break small connections using morphological open */
element = AVW_CreateStructuringVolume(AVW_26_CONNECTED,  3, 3, 3, NULL);
volume3 = AVW_MorphOpenVolume(volume2, element, NULL);
volume3 = AVW_ConditionalDilateVolume(volume3, volume2, element, volume3);

                                           /* Find Connected Components */
volume1 = AVW_FindVolumeComponents(volume3, AVW_SORT_AND_LABEL,
                   AVW_26_CONNECTED, volume3->VoxelsPerVolume, 1, volume1);

                                /* Threshold to isolate largest component */
volume1 = AVW_ThresholdVolume(volume1, 1.0, 1.0, volume1);
```

CHAPTER 8

Analysis

Images and volumes contain a wealth of information from which specific measurements can be made. **AVW** provides routines to compute line profiles and histograms and measure intensity and shape statistics. The measurements may be performed on the entire image or volume or, more commonly, on a user specified subregion of the image or volume which can be represented by a mask. Structures and prototypes relating directly to the measurement routines are located in the **AVW_Measure.h** include file.

8.1 Line Profiles

AVW_ComputeLineProfile() takes an image and a user supplied trace, interpolates the points of a trace to make a continuous segment, and returns an **AVW_PointValueList** which contains the X and Y coordinates of the interpolated trace along with the corresponding gray scale values from the image. The length of the trace is also returned through the parameter list. In this manner, the user can pass the endpoints of a line or the points of a user defined trace, to the routine and get the corresponding gray scale values.

FIGURE 8-1 **Line Profile Example**

```
AVW_PointList2 *trace;
AVW_PointValueList *profile=NULL;
AVW *image;
AVW_Point2 point;
double length;
     .
     .
     .
trace = AVW_CreatePointList2(2);
point.X = 0;
point.Y = image->Height/2;
```

```
AVW_AddPoint2(trace, &point);
point.X = image->Width-1;
point.Y = image->Height/2;
AVW_AddPoint2(trace, &point);
profile = AVW_ComputeLineProfile(image, trace, &length);
```

8.2 Masks

A mask is an image or volume in which pixels or voxels with the same value define distinct, connected or unconnected, regions. The mask is used in conjunction with the gray scale image or volume. When using a mask, pixels from the gray scale image or volume are sampled only if the corresponding pixel or voxel in the mask is equal to a user specified **`mask_value`**. If the mask is set to **`NULL`**, the entire image or volume will be sampled and the **`mask_value`** is ignored. In the simple case, the mask can be a thresholded image or volume. The thresholding operation sets an image or volume to ones and zeros. The ones define a mask and these pixels and their related gray scale values can be sampled separately. Masks may also be interactively defined by a user's trace. The following routine creates a mask from a trace. An image with all of the pixels inside the trace set to one, and all of the pixels outside the trace set to zero, is created. The pixels under the trace can be set to zero or one based on the **`under_border`** parameter.

FIGURE 8-2 **Make Mask Example**

```
AVW_Image *mask_image = NULL;
AVW_PointList2 *trace=NULL;
AVW_Point2 point_inside_trace;
int under_border = 1;
int width, height;

trace = AVW_CreatePointList2(256);
     .                             /* Get user trace, AVW_AddPoint2() loop*/
     .            /* Set point_inside_trace.X and point_inside_trace.Y to */
     .                              /*     average of all points in trace */
     .                    /* Set width and height to desired size of mask */

mask_image = AVW_MakeMaskFromTrace(trace, &point_inside_trace, width,
                                               height, under_border, NULL);
```

8.3 Histograms

A histogram describes the gray level distribution of an image or volume. The **AVW_Histogram** structure, defined in **AVW_Histogram.h**, contains the necessary elements to define a histogram for any data type.

```
typedef struct
    {
    double *Mem;
    double Max;
    double Min;
    double Step;
    unsigned int Bins;              /* Number of bins in the histogram */
    } AVW_Histogram;
```

The **Mem** element is a pointer to an array of bins which contains the number of occurrences of each intensity within the image or volume. **Mem[0]** contains the number of occurrences of the **Min** intensity. The **Max** and **Min** elements define the range of the histogram and **Step** defines the width of each bin. **AVW** has routines for creating, clearing and collecting histograms. Processing of images or volumes using histogram operations are also supported (See Chapter 5). Histograms may be collected from a user specified region by using a mask. If the mask is **NULL** the histogram is collected from the entire image or volume. Histograms may be reused in the same manner as images or volumes. Note that if a **NULL** pointer to a histogram is passed to a routine, the histogram will be created with the **Max** and **Min** elements of the histogram set to the maximum and minimum values of the image or volume being sampled.

FIGURE 8-3 **Histogram Collection**

```
#include “AVW_Histogram.h”
AVW_Volume *volume;
AVW_Image *image, *mask_image;
AVW_Histogram *histogram;
int i;
    .
    .
    .

histogram = AVW_GetVolumeHistogram(volume, NULL, 0, NULL);
if(histogram == NULL)
```

```
{
    AVW_Error(“AVW_GetVolumeHistogram”);
}
else
{
    for(i=0; i < histogram->Bins; i++)
       fprintf(stdout, “%d:%d  “, i,(int)histogram->Mem[i]);
    fprintf(stdout,"\n\n");
}
.
mask_image = AVW_ThresholdImage(image, 255.0, 120.0, NULL);
histogram = AVW_GetImageHistogram(image, mask_image, 1, NULL);
if(histogram == NULL)
{
    AVW_Error(“AVW_GetVolumeHistogram”);
}
else
{
    for(i=0; i < histogram->Bins; i++)
       fprintf(stdout, “%d:%d  “,i,(int)histogram->Mem[i]);
    fprintf(stdout,"\n");
}
```

`AVW_GetImageHistogram()` and **`AVW_GetVolumeHistogram()`** create the histogram if necessary and set each bin to zero before sampling the image or volume. Other histogram related routines include: **`AVW_ClearHistogram()`** which sets the count in each bin of the histogram to zero and **`AVW_CreateHistogram()`** which creates a histogram according to user specified parameters.

8.4 Intensity Statistics

Intensity measurements can be obtained from images, volumes, subimages and subvolumes which are defined with a mask, and a specified range of gray scale values within an image or volume as illustrated in the following sample code.

FIGURE 8-4 **Intensity Statistics Example**

```
#include “AVW_Measure.h”
AVW_Volume *volume, *mask_volume=NULL;
AVW_Image *image, *mask_image = NULL;
AVW_IntensityStats stats;
int check, voxels;
double sample_max, sample_min;
int mask_value = 1;
.
.
.
sample_max = 255.0; /* Max and Min for 8 bit data */
sample_min = 0.0;
                        /* Sample an entire image (mask_image = NULL)*/
check = AVW_ComputeImageIntensityStats(image, mask_image,
                       mask_value, sample_max, sample_min, stats);
if(check == AVW_FAIL)
{
    AVW_Error(“AVW_ComputeImageIntensityStats”);
}
else
{
    printstats(stats);
}
                               /* Sample a masked region of an image */
mask_image = AVW_ThresholdImage(image, 255.0, 120.0, NULL);
check = AVW_ComputeImageIntensityStats(image, mask_image, mask_value,
                 sample_max, sample_min, stats);
if(check == AVW_FAIL)
{
    AVW_Error(“AVW_ComputeImageIntensityStats”);
}
else
{
    printstats(stats);
}
```

```
                         /* Sample an entire volume  (mask_volume = NULL)*/
     check = AVW_ComputeVolumeIntensityStats(volume, mask_volume,
                     mask_value, sample_max, sample_min, stats);
     if(check == AVW_FAIL)
     {
         AVW_Error(“AVW_ComputeVolumeIntensityStats”);
     }
     else
     {
         printstats(stats);
     }
/* Sample a gray scale range of a volume  (mask_volume = NULL)*/
     sample_max = 200.0;
     sample_min = 100.0;
     check = AVW_ComputeVolumeIntensityStats(volume, mask_volume,
mask_value, sample_max, sample_min, stats);
     if(check == AVW_FAIL)
     {
         AVW_Error(“AVW_ComputeVolumeIntensityStats”);
     }
     else
     {
         printstats(stats);
     }
```

The **`mask_image`**, **`mask_volume`** and **`mask_value`** parameters allow the user to sample a subregion of an image or volume. The **`sample_max`** and **`sample_min`** parameters allow the user to specify a gray scale range on which separate statistics will be calculated. The functions return **`AVW_SUCCESS`** if the no error occurred. If an error was encountered, **`AVW_FAIL`** is returned and **`AVW_ErrorNumber`** and **`AVW_ErrorMessage`** are set to values indicating the type of error.The intensity statistics are stored in the provided **`AVW_IntensityStats`** structure, **`stats`**, which is defined in **`AVW_Measure.h`** as follows.

FIGURE 8-5 **AVW_IntensityStats Structure**

```
typedef struct
{
double Mean;
double StandardDeviation;
double Variance;
double Sum;
double SumOfSquares;
unsigned long NumberOfVoxels;
double Area;
double Volume;
double HighestIntensity;
AVW_Point3 HighestPoint;
double LowestIntensity;
AVW_Point3 LowestPoint;
double RangeMaximum;
double RangeMinimum;
double MeanInRange;
double StandardDeviationInRange;
double VairianceInRange;
double SumInRange;
double SumOfSquaresInRange;
unsigned long NumberBelowRange;
unsigned long NumberInRange;
unsigned long NumberAboveRange;
double BrightnessAreaProduct;
} AVW_IntensityStats;
```

The following numeric values are used from the **`Info`** string in the **`AVW_Image`** or **`AVW_Volume`** (See **`AVW_PutNumericInfo()`**): **`VoxelWidth`**, **`VoxelHeight`**, **`VoxelDepth`** and **`SliceNumber`**. The string value for **`Orientation`** (Transverse, Coronal, or Sagittal) is also used (See **`AVW_PutStringInfo()`**). Most of the elements of this structure are self-explanatory. The **`Area`** is the **`NumberOfVoxels`** multiplied by the **`VoxelHeight`** and **`VoxelWidth`**. The **`Volume`** is computed by multiplying the **`Area`** by the **`VoxelDepth`**. **`HighestIntensity`** is the highest gray level value of the pixels which were sampled. **`HighestPoint`** gives the location of the first occurrence of the **`HighestIntensity`**.

`LowestIntensity` and `LowestPoint` are similarly defined for the lowest gray level value sampled. `RangeMaximum` and `RangeMinimum` are set to the `sample_max` and `sample_min` parameters passed to the function. The mean, standard deviation, variance, sum, and sum of squares are computed on voxels whose gray levesl are within the specified range. `NumberBelowRange` specifies the number of sampled pixels whose gray level value was less than the `RangeMinimum`. `NumberInRange` specifies the number of sampled pixels whose gray level was greater than or equal to the `RangeMinimum` and less than or equal to the `RangeMaximum`. The `NumberAboveRange` specifies the number of sampled pixels whose gray level value was greater than the `RangeMaximum.` The `BAP` (Brightness Area Product) is calculated by subtracting the `RangeMinimum` from each pixel value and then summing all of the pixel values which are greater than zero.

8.5 Shape Statistics

2D shape measurements can be obtained from a masked region within an image. The shape statistics are stored in the provided **AVW_2DShapeStats** structure which is defined in **AVW_Measure.h** as follows.

FIGURE 8-6 **AVW_2DShapeStats Structure**

```
typedef struct
    {
    float   Area;
    float Perimeter;
    float MERAngle;       /*Minimum Eclosing Rectangle Angle in degrees */
    float MERArea;
    float MERAspect;
    float RFF;                                /* Rectangle Fit Factor */
    float Circularity;
    AVW_FPoint2 Centroid;
    } AVW_2DShapeStats;
```

The **Area** is the count of all pixels in the **mask_image** equal to the **mask_value**. **Perimeter** is the distance around a piecewise linear closed curve constructed between the outermost pixel of the region defined in the **mask_image**. **MER** stands for **M**inimum **E**nclosing **R**ectangle and three statistics about the **MER** that surrounds the object defined in the **mask_image** are reported: **MERAngle** is the angle of orientation, in degrees, of the **MER**, **MERArea** is the width of the **MER** multiplied by the height, and **MERAspect** is the aspect ratio of the **MER** (short side divided by the long side). **RFF** is the **R**ectangular **F**it **F**actor which is the ratio of the area of the region to the area of its **MER**. **Circularity** is the ratio of the perimeter of the region squared to its area. It takes on a minimum value of 4*pi for a disk and higher values for more complex shapes. **Centroid** is the average of all of the x and y coordinates of all the pixels in the region defined in the **mask_image**.

FIGURE 8-7 **Shape Measurement Example**

```
#include “AVW_Measure.h”
AVW_Image *image, *mask_image;
int mask_value = 1, check;
AVW_2DShapeStats stats;
     .
     .
     .

     mask_image = AVW_ThresholdImage(image, 255.0, 220.0, NULL);
     check = AVW_Compute2DShapeStats(mask_image, mask_value, &stats);
     if(check == AVW_FAIL)
     {
          AVW_Error(“AVW_Compute2DShapeStats”);
     }
     else
     {
          printshapestats(stats);
     }
```

The **mask_image** and **mask_value** parameters specify a segmented region within an image on which the shape statistics are to be collected. The functions return **AVW_SUCCESS** if the no error occurred. If an error was encountered, **AVW_FAIL** is returned and **AVW_ErrorNumber** and **AVW_ErrorMessage** are set to values indicating the type of error. **AVW** also has routines that support the collection of these shape statistics individually. See the manual pages for **AVW_ComputeCircularity()**, **AVW_ComputeImageCentroid()**, **AVW_ComputeMER()**, **AVW_ComputePerimeter()** and **AVW_ComputeRFF()** for more details.

AVW also has three routines which perform 3D shape measurements. The **MEB** (**M**inimum **E**nclosing **B**rick), volume and centroid can be computed by calling AVW routines.

FIGURE 8-8 **3D Measurement Example**

```
#include “AVW_Measure.h”
AVW_Volume *volume, *mask_volume;
double number_of_voxels, angles[3], extents[3];
int check, voxels;
int mask_value = 1;
AVW_FPoint3 centroid;
     .
     .
     .
mask_volume = AVW_ThresholdVolume(volume, 255.0, 120.0, NULL);
check = AVW_ComputeMEB(mask_volume, mask_value, 5.0, &number_of_voxels,
                                                          angles, extents);
if(check == AVW_FAIL)
{
    AVW_Error(“AVW_ComputeMEB”);
}
else
{
    fprintf(stdout, “Number of Voxels = %f\n”,number_of_voxels);
    fprintf(stdout, “MEB Orientation Angles (X, Y, Z) = %f, %f, %f\n”,
                                          angles[0], angles[1], angles[2]);
    fprintf(stdout, “MEB Sizes (X, Y, Z) = %f, %f, %f\n”, extents[0],
                                                   extents[1], extents[2]);
}
voxels = AVW_ComputeVolume(mask_volume, 1);
fprintf(stdout, “Number of Voxels = %d\n”,voxels);
check = AVW_ComputeVolumeCentroid(mask_volume, mask_value, &centroid,
                                                                  &voxels);
if(check == AVW_FAIL)
{
    AVW_Error(“AVW_ComputeVolumeCentroid”);
}
else
{
    fprintf(stdout, “Number of Voxels = %d\n”,voxels);
    fprintf(stdout, “Centroid (X, Y, Z) = %f, %f, %f\n”,centroid.X,
                                                    centroid.Y, centroid.Z);
}
```

`AVW_ComputeMEB()` calculates the **M**inimum **E**nclosing **B**rick around the voxels equal to the **`mask_value`** in the **`mask_volume`**. A rotation step, in degrees, which is used to search for the **`MEB`** is specified by the user. The function returns through the parameter list: the number of voxels contained by the **`MEB`**, the rotation angles on the x, y and z axes of the **`MEB`**, and the x, y and z dimensions of the **`MEB`**. **`AVW_ComputeVolume()`** returns a count of the voxels in the **`mask_volume`** which are equal to the **`mask_value`**.
`AVW_ComputeVolumeCentroid()` computes the centroid by averaging the x and y coordinates of each voxel equal to the **`mask_value`** in the **`mask_volume`**.

8.6 Sample Code

An example program using the measurement routines described in this chapter is located in **`$AVW/extras`**.

CHAPTER 9

Visualization

Volume rendering techniques based on ray-casting algorithms have become the method of choice for the visualization of 3D biomedical volumes. These methods provide direct visualization of the volumes without the need for prior surface or object segmentation, preserving the values and context of the original image data. Volume rendering techniques allow for the application of various different rendering algorithms during the ray-casting process.

9.1 Render Parameters

The ray-casting function within **AVW**, **AVW_RenderVolume()**, has the capability to do many things. Much of this power is accomplished by using many user controlled parameters. To simplify parameter passing, each rendering parameter is a member of the **AVW_RenderParameters** structure.

FIGURE 9-1 **AVW_RenderParameters Structure**

```
typedef struct
              {
              int Type;
              double ThresholdMinimum, ThresholdMaximum;
              int ClipLowX, ClipLowY, ClipLowZ;
              int ClipHighX, ClipHighY, ClipHighZ;
              int ClipPlaneMinimum, ClipPlaneMaximum;
              int ClipShading;
              int RenderWidth, RenderHeight, RenderDepth;
              int MaximumPixelValue, MinimumPixelValue;
              int SurfaceThickness;
              AVW_Matrix *Matrix;
```

```
                AVW_Matrix *LightMatrix;
                AVW_Image *RenderMask;
                int MaskValue;
                int DeleteDepth;
                double DeleteValue;
                double ScaleX, ScaleY, ScaleZ;
                int PerspectiveType;
                AVW_FPoint3 EyePosition;
                double XFieldOfViewAngle, YFieldOfViewAngle;
                double SpecularFactor;
                double SpecularExponent;
                double ReservedForFuture1;
                double ReservedForFuture2;
                double ReservedForFuture3;
                double ReservedForFuture4;
                AVW_InternalParameters Internal;
                } AVW_RenderParameters;
```

AVW provides an initialization function which initializes all the rendering parameters to reasonable values and allows the user to focus only on the parameters needed to accomplish the user specific task.

FIGURE 9-2 Initialization Example

```
     AVW_Volume *volume = NULL;
     AVW_RenderParameters *render_param = NULL;
     AVW_RenderedImage *rendered = NULL;
       .
       .
       .
     render_param = AVW_InitializeRenderParameters(volume, NULL,
render_param);
       .
       .
       .
     if((rendered = AVW_RenderVolume(render_param, rendered)) == NULL)
     {
           AVW_Error("AVW_RenderVolume");
           break;
     }
       .
       .
       .
```

9.1.1 Render Type

The **`Type`** parameter specifies which ray casting algorithm should be used. The available options are shown in Figure 9-3 and described in Table 9-1.

FIGURE 9-3 **Render Types Defined in AVW_Render.h**

```
#define AVW_DEPTH_SHADING              0
#define AVW_GRADIENT_SHADING           1
#define AVW_MAX_INTENSITY_PROJECTION   2
#define AVW_SUMMED_VOXEL_PROJECTION    3
#define AVW_SURFACE_PROJECTION         4
#define AVW_TRANSPARENCY_SHADING       5
#define AVW_DELETE_VOXELS              6
```

TABLE 9-1 Render Type Algorithms

Type	Description
AVW_DEPTH_SHADING	The value of the display pixel is a function of depth only (distance between the screen and the intersected voxel on the surface). The depth at which the first voxel found is used to determine the initial brightness of the pixel in the shaded surface image. Near voxels appear brighter than distant voxels. The **`AVW_ProcessZGradients()`** function can be used to enhance a depth shaded rendering by highlighting changes in neighboring rendered pixels.
AVW_GRADIENT_SHADING	The gray scale gradient vector is computed using a 3-D neighborhood about the Surface Voxel. The value projected to the rendered image is the dot product of the gradient vector and an independently specified light source.This simulates the appearance of a reflective surface under uniform-field illumination.
AVW_MAX_INTENSITY_PROJECTION	The value of the voxel with maximal value (within the threshold criteria) along the ray is placed into each pixel of the rendered image. If no voxel in the threshold range exists on the ray the **`MinimumPixelValue`** is returned.
AVW_SUMMED_VOXEL_PROJECTION	The rendered pixel is computed as an average of all voxels along the ray falling in a specified threshold range.
AVW_SURFACE_PROJECTION	The rendered pixel is computed as an average of the first N voxels along the ray path. The count does not begin until the first voxel within the threshold criteria is encountered. The value of N is specified with the **SurfaceThickness** parameter.

TABLE 9-1 Render Type Algorithms

Type	Description
AVW_TRANSPARENCY_SHADING	This algorithm is only available when an object map has been specified on the call to **`AVW_InitializeRenderParameters()`**. This algorithm uses the **`Opacity`** and **`OpacityThickness`** parameters of each of the objects along the ray path to produce a 24-bit color rendering. See the chapter on Object Maps for more information.
AVW_DELETE_VOXELS	Instead of casting rays to produce a rendered image, this algorithm is used to delete voxels along the ray path. If an object map was specified during initialization, it can also be used to define voxels in an object map. The **`DeleteDepth`** parameter specifies the action (See Figure 9.4) which occurs during the raycasting process. The "`DELETE`" actions cause voxels along the ray path, which meet the rendering criteria, to be changed to the value specified in **`DeleteValue`**. The"`DEFINE`" actions (only when object map has been specified) cause voxels in the object map to be changed to the value specified by **`DeleteValue`**. A **`DeleteDepth`** of **`ALL_THE_WAY`** causes all voxels along the ray path to be deleted or redefined. The **`SINGLE_LAYER`** actions cause the delete or redefine action to cease once the first object contacted is exited. The **`SINGLE_VOXEL`** actions cause only the first voxel meeting the rendering criteria to be deleted or redefined. NOTE: This algorithm will most likely be used with the **`RenderMask`** and **`MaskValue`** parameters to specify a limited portion of the rendered image to modify. The render call which does the delete/redefine will probably be followed by another render call using one of the other render types, which will use the same **`RenderMask`** to re-render the deleted/redefined area.

FIGURE 9-4 Delete Depth Definitions in AVW_Render.h

```
#define AVW_DELETE_ALL_THE_WAY         0
#define AVW_DELETE_SINGLE_LAYER        1
#define AVW_DELETE_SINGLE_VOXEL        2
#define AVW_DEFINE_ALL_THE_WAY         3
#define AVW_DEFINE_SINGLE_LAYER        4
#define AVW_DEFINE_SINGLE_VOXEL        5
```

9.1.2 Thresholds

The **`ThresholdMinimum`** and **`ThresholdMaximum`** parameters specify which voxels along the ray path will be processed. Only voxels within this range will be used in determining surface voxel, maximum voxels or during the summing process.

9.1.3 Translation and Rotation

The **`Matrix`** parameter is used to specify any combination of translation or rotation which is to be applied to the input volume and object map during the rendering process.

FIGURE 9-5 **360 Degree Rotation Sequence**

```
AVW_Volume *volume = NULL;
AVW_RenderParameters *render_param = NULL;
AVW_RenderedImage *rendered = NULL;
double angle, increment = 3.;
       .
       .
       .
render_param = AVW_InitializeRenderParameters(volume, NULL,
render_param);

AVW_SetIdentityMatrix(render_param->Matrix);  /* Not really needed */
AVW_RotateMatrix(render_param->Matrix, -90., 0., 0.);/* Rotate so  */
     /* head faces forward */
for(angle=0; angle < 360; angle += increment)
{
     AVW_RotateMatrix(render_param->Matrix, 0., increment, 0.);
     if((rendered = AVW_RenderVolume(render_param, rendered)) == NULL)
     {
          AVW_Error("AVW_RenderVolume");
          break;
     }
}
```

9.1.4 Scale

The **`ScaleX, ScaleY,`** and **`ScaleZ`** parameters specify scale factors for voxel width, height, and depth respectively. A value of 1.0 indicates no scaling.

9.1.5 Clip Volume

The **`ClipLowX`**, **`ClipLowY`**, **`ClipLowZ`**, **`ClipHighX`**, **`ClipHighY`** and **`ClipHighZ`** parameters specify the 3D region of the volume to be rendered.

9.1.6 Clip Plane

The **`ClipPlaneMinimum`** and **`ClipPlaneMaximum`** parameters specify starting and ending rendering planes which are perpendicular to the ray casting rays specified by the **`Matrix`** parameter. Voxels which occur along the casting ray which are before the **`ClipPlaneMinimum`** or after the **`ClipPlaneMaximum`** are ignored. These values range from 0 to **`RenderDepth`** (square root of the sum of the squared volume dimensions). 0 being the closest possible plane and **`RenderDepth`** the farthest possible plane.

9.1.7 Clip Shading

The **`ClipShading`** parameter specifies the shading type used when the rendering process starts inside a volumes object. Possible values:

`AVW_CLIP_SHADED`- Calculate a shading value based only on the voxels distance along the ray path. Closer voxels will be brighter than distant voxels.

`AVW_CLIP_ACTUAL` - Use the actual value of the current voxel.

`AVW_CLIP_REMOVE_AND_RENDER` - Remove the current voxel and then calculate the gradient shading value.

`AVW_CLIP_RENDER_AS_IS` - Calculate the gradient shading value without any other processing.

9.1.8 Render Buffer Dimensions

The **`RenderWidth`** and **`RenderHeight`** parameters specify the size of the rendered image to be generated. The **`RenderDepth`** specifies the maximum range of values in the **`ZBuffer`** and **`PBuffer`**. These values default to the square root of the sum of the squares of each of the dimensions of the volume.

9.1.9 Render Values

The **`MaximumPixelValue`** and **`MinimumPixelValue`** parameters specify the range of legal values in the rendered image.

9.1.10Lighting

The **`LightMatrix`** parameter in an **`AVW_Matrix`** which specifies the direction of the current light source. The default is an identity matrix, which specifies the light source is always the same as the view point specified by the **`Matrix`** parameter discussed earlier.

9.1.11Render Mask

The **`RenderMask`** is a pointer to an **`AVW_Image`** which specifies an area to be processed. The **`Render-Mask`** image should always have a size of **`RenderWidth`** by **`RenderHeight`**. Any voxel in the **`Ren-derMask`** which has a value of **`MaskValue`** causes a ray to be recast for that voxel in the rendered image. The default is **`NULL`**, which indicates the entire image is to be processed.

9.1.12 Perspective Rendering

The **`PerspectiveType`** parameter specifies the type of rendering to be performed parallel or perspective. **`AVW_PERSPECTIVE_INT`** renders the image using the voxels without interpolation, possibly resulting in “blocky” renderings. If **`PerspectiveType`** is set to **`AVW_PERSPECTIVE_FLOAT`**, the rendered image is generated using an on the fly interpolation rendering algorithm. If **`PerspectiveType`** is set to **`AVW_PERSPECTIVE_OFF`**, parallel rendering is performed. For perspective rendering, the render matrix is interpreted as a viewing direction for the camera model, however the parallel rendering conventions are followed, i.e. the matrix is left handed, and the identity matrix provides a view along the positive Z axis with X increasing to the right and Y increasing in the vertical direction. Perspective rendering also uses the **`EyePosition`** parameter when generating the rendering, thus the viewing direction and the position must be coordinated to produce the proper view. The default values assigned by the **`AVW_InitializeRenderParameters()`** call will produce a rendering of the X-Z plane from positive Y with a viewing direction along Y, and Z as the up direction, with the **`XFieldOfViewAngle`** and **`YFieldOfViewAngle`** set to the approximate size of the default parallel rendering. Perspective rendering does not support scaling at this time. For anisotropic data, rescaling during loading is the best option.

The **`EyePosition`** parameter specifies the location of the camera model used to generate the rendering. This parameter is only used when **`PerspectiveType`** is set to **`AVW_PERSPECTIVE_FLOAT`** or **`AVW_PERSPECTIVE_INT`**. The X, Y, and Z coordinates of the **`AVW_FPoint3`** structure refer to the position of the camera relative to the center of the volume, thus a position of (0,0,0) in a particular volume is located at voxel (Width / 2, Height / 2, Depth / 2). The Eye position is not constrained in any manner; the camera model may be located inside the volume (in effect, virtual endoscopy) or arbitrarily removed from the volume to simulate a parallel rendering.

The **`XFieldOfViewAngle`** and **`YFieldOfViewAngle`** parameters specify the field of view (FOV) angle of the camera model used to generate the image. Increasing the FOV results in a zoom out effect, if the position remains the same. Decreasing the FOV results in a zoom in effect.

9.1.13 Specular Reflection

The **`SpecularFactor`** parameter specifies the ratio of gradient shading to specular shading. If **`SpecularFactor`** is set to 0, the rendering will be shaded entirely using gradient shading with no performance penalty. If **`SpecularFactor`** is set to 1, the rendering is shaded entirely using the specular shading model.

The **`SpecularExponent`** parameter specifies the degree of fall-off for specular shading. High values (about 10) result in very small specular highlights, while small values (about 1 or 2) result in diffuse highlights.

9.2 Post Processing

The **`AVW_ProcessZGradients()`** function can be used to enhance a **`DepthShaded`** image. The **`AVW_MirrorRendered()`** function can be used to mirror on a specified axis.

9.3 Interface

When building tools to interact with a rendered image, the **`AVW_FindRenderedPoint()`** can be used to convert a 2D location within a rendered image, back to the original 3D location in the input volume.

FIGURE 9-6 Screen Coordinates to Render Space

```
AVW_RenderParameters *render_param = NULL;
AVW_RenderedImage *rendered = NULL;
AVW_Point2 point2d;
AVW_Point3 point3d;
       .
       .
if((rendered = AVW_RenderVolume(render_param, rendered)) == NULL)
```

```
{
    AVW_Error("AVW_RenderVolume");
    break;
}

point2d.X = rendered->Image.Width / 2;/* In a real working example */
point2d.Y = rendered->Image.Height / 2;   /* this point would probably be
*/
                                         /* selected with the mouse
*/

if(!AVW_FindRenderedPoint(rendered, &point2d, &point3d))
{
    printf("No rendered voxel in middle of the image\n");
}
    else
{
    printf("(%dx%dx%d) = %d\n", point3d.X, point3d.Y, point3D.Z,
    (int) AVW_GetVoxel(rendered->Volume, &point3d) );

}
```

The **`AVW_DrawRenderedLine()`** and **`AVW_DrawRenderedPoint()`** functions can be used to draw graphically into a rendered image. The input values are specified in the input volumes 3D space.

9.4 Interactive Surfaces

The **`AVW`** rendering engine is very fast, but even it can't keep up with the demands of interactive movements. To assist in producing even faster renderings, **`AVW`** provides a function which extracts all the visible surface points, **`AVW_ExtractVisibleSurface()`**, and another routine to quickly render these points at another user specified angle, **`AVW_RenderVisibleSurface()`**. This only works when the render type is **`GradientShading`** and each change of the threshold range will require re-extraction of the surface. The **`AVW_DestroyVisibleSurface()`** can be used to clean-up extracted surfaces.

9.5 Clean Up

When finished with a rendered image, the memory associated can easily be freed with a call to **`AVW_DestroyRenderedImage()`**. When no further renderings are to be done with a volume, the parameter structure can easily be freed with **`AVW_DestroyRenderParameters()`**.

CHAPTER 10

Object Maps

An object map, when associated with an **AVW_Volume**, allows attributes to be assigned to a voxel or group of voxels. An object map must have the same dimensions as the associated volume and will always be of data type 8-bit unsigned char.

10.1 AVW_ObjectMap Structure

AVW uses the **AVW_ObjectMap** structure (See Figure 10-1) to pass object map information to and from functions. Object maps were first used in the 1988 version of Volume Render in ANALYZE'. Since then, the format has changed many times.

FIGURE 10-1 **AVW_ObjectMap structure defined in AVW_ObjectMap.h**

```
typedef struct
     {
     int Version;
     int NumberOfObjects;
     AVW_Volume *Volume;
     AVW_Object *Object[256];
     unsigned char ShowObject[256];
     unsigned char MinimumPixelValue[256];
     unsigned char MaximumPixelValue[256];
     } AVW_ObjectMap;
```

The **`Version`** member of **`AVW_ObjectMap`** is used to indicate which variation of information is contained in the **`AVW_Object`** structure. This is more important when the information is stored on disk. The **`AVW_LoadObjectMap()`** function can read all the formats that have existed over the years, defaulting values that may have been added. **`AVW_SaveObjectMap()`** writes only the latest version.

The **`NumberOfObjects`** member indicates the current number of objects defined. This value should never be less than 1, and in theory could be as high as 256, but problems with color usage limits this to a much smaller value (See **`Shades`** in the **`AVW_Object`** description).

The **`Volume`** member is an **`AVW_Volume`** which contains the actual map of which voxels belongs to which object. A value of zero indicates that the voxel in the associated with the first object (**`Object[0]`**), a value of one, the second object (**`Object[1]`**), etc... Do not confuse this **`AVW_Volume`** with the volume which it is associated with.

The **`Object`** member is an array of 256 pointers to **`AVW_Object`** structures. Only the first N (N = **`NumberOfObjects`**) are defined, the others are should be **`NULL`**.

The **`ShowObject`**, **`MinimumPixelValue`** and **`MaximumPixelValue`** are arrays used internally. The values in them should not modified or used.

10.2 AVW_Object Structure

The **`AVW_Object`** structure (See Figure 10-2) contains the attributes to be associated with each object. Each object in an **`AVW_ObjectMap`** has an **`AVW_Object`** structure defined for it. An **`AVW_Object`** contains fields which are used in the **`ANALYZE`** Volume Render module, but have not yet been implemented into the **`AVW_RenderVolume()`** function, these values are included for backward compatibility. In a future, more powerful, version of **`AVW_RenderVolume()`**, it is expected that these capabilities will once again be supported and even expanded.

FIGURE 10-2 AVW_Object Structure

```
typedef struct
{
    char Name[32];
    int DisplayFlag;
    unsigned char CopyFlag;
    unsigned char MirrorFlag;
    unsigned char StatusFlag;
    unsigned char NeighborsUsedFlag;
    int Shades;
    int StartRed, StartGreen, StartBlue;
    int EndRed, EndGreen, EndBlue;
    int XRotation, YRotation, ZRotation;
    int XTranslation, YTranslation, ZTranslation;
    int XCenter, YCenter, ZCenter;
    int XRotationIncrement, YRotationIncrement, ZRotationIncrement;
    int XTranslationIncrement;
    int YTranslationIncrement;
    int ZTranslationIncrement;
    short int MinimumXValue, MinimumYValue, MinimumZValue;
    short int MaximumXValue, MaximumYValue, MaximumZValue;
    float Opacity;
    int OpacityThickness;
    int Dummy;
} AVW_Object;
```

The **`Name`** member of the **`AVW_Object`** structure contains a user defined name for this object. Any zero-terminated string can be used, including strings with embedded spaces.

`DisplayFlag` is used to enable or disable the display of this particular object. A value of zero value indicates that voxels of this object type are ignored or assumed to be outside the threshold range during the raycasting process.

`CopyFlag` indicates object translation, rotation, and mirroring are applied to a copy of the actual object, rather than the object itself. [**`ANALYZE`** only]

`MirrorFlag` indicates the axis this object is mirrored around. [**`ANALYZE`** only]

`StatusFlag` is used to indicate when an object has changed and may need it's minimum bounding box recomputed. [**`ANALYZE`** only]

`NeighborsUsedFlag` indicates which neighboring voxels are used in calculating the objects shading. [**`ANALYZE`** only]

`Shades` indicates the number of shades available for this object. Only 256 (250 in **`ANALYZE`**) total shades are available.

`StartRed`, **`StartGreen`**, and **`StartBlue`** specify the starting color for this object. This is usually a darker shade of the ending color. **`ANALYZE`** defaults these values to 10% of the ending color.

`EndRed`, **`EndGreen`**, and **`EndBlue`** specify the ending color for this object.

XRotation, **YRotation**, and **ZRotation** specify a rotation which is applied to this object only. [**ANALYZE** only]

XTranslation, **YTranslation**, and **ZTranslation** specify a translation which is applied to this object only. [**ANALYZE** only]

XCenter, **YCenter**, and **ZCenter** specify the rotation center, relative to the volumes center, which the **XRotation**, **YRotation**, and **ZRotation** are rotated around. [**ANALYZE** only]

XRotationIncrement, **YRotationIncrement**, and **ZRotationIncrement** specify increments that are applies to **XRotation**, **YRotation**, and **ZRotation** when making a sequence. [**ANALYZE** only]

XTranslationIncrement, **YTranslationIncrement**, and **ZTranslationIncrement** specify increments that are applies to **XTranslation**, **YTranslation**, and **ZTranslation** when making a sequence. [**ANALYZE** only]

MinimumXValue, **MinimumYValue**, **MinimumZValue**, **MaximumXValue**, **MaximumYValue**, and **MaximumZValue** specify the minimum enclosing brick used by **ANALYZE** to increase the rendering speed of individual objects during object translations and rotations. [**ANALYZE** only]

Opacity and **OpacityThickness** are used only when rendering 24-bit transparency images. **Opacity** is a floating point value between .0001 (very transparent) and 1.0 (opaque). The **Opacity** value is multiplied times the color of each surface voxel intersected, it is also subtracted from 1.0 to determine the amount of shading which can be applied by objects intersected after this object. Ray-casting continues as long as it is possible for remaining objects along the ray path to make contributions. The **OpacityThickness** parameter allows the surface determined color, to be added in multiple times for thicker objects. This allows the thickness of each object to make a contribution into what can be seen behind it. A value of 1 for **OpacityThickness** results in only the surface of each object having a contribution.

Dummy is not used, but included for possible future expansion.

10.3 Creation and Destruction of Object Maps

The **AVW_CreateObjectMap()** function can be used to create an object map. The dimensions of the volume being created for the object map for must be supplied to the **AVW_CreateObjectMap()** call. The created object map contains only one object (**Object[0]**) and the Volume, which is used for mapping, contains all zeros. The **AVW_DestroyObjectMap()** function is used to destroy all memory associated with an object map. See Figure 10.3.

FIGURE 10-3 **Creating and Deleting an Object Map**

```
AVW_Volume *volume;
AVW_ObjectMap *object_map;
    object_map = AVW_CreateObjectMap(volume->Width, volume->Height,
                                     volume->Depth);
  .
  .
  .
    AVW_DestroyObjectMap(object_map);
    object_map = NULL;
```

10.4 Saving an Object Map to Disk

`AVW_SaveObjectMap()` is used to save all information contained in an object map to disk (Figure 10.5 shows the disk format). All object map files have an **`.obj`** suffix, if this is not included on the file name specified it will automatically be appended. See Figure 10.4.

FIGURE 10-4 **Saving an Object Map**

```
AVW_ObjectMap *object_map;
char filepath[256];
  .
  .
  .
AVW_SaveObjectMap(filepath, object_map);
```

10.5 Object Map File Format

The object map file (.obj extension) consists of header information describing the current number of objects and each object's parameters, followed by the values which comprise the object map itself. Each value may hold an 8-bit unsigned value, allowing for the definition of 256 total objects. There is a one-to-one correspondence between the voxels in the volume image and the “voxels” in the object map. The object map is run length encoded to take advantage of the fact that many objects are spatially connected, allowing for good compression of the file.

The header format consists of an initial set of information describing the total object map (Fields A-E below), a set of information that is object specific (Fields F1-F34 below) which is repeated in the file for each of current objects (i.e. F1-F34 for Object 1 followed by F1-F34 for Object 2 followed by ...), and then followed by the object map values themselves. The following describes each parameter stored in the object information:

TABLE 10-1

Field	Name	Type	Description
A	version	int	Six different revisions of the **ANALYZE** .obj format have existed since late '87 when the format was defined. The current version is 910926, appeared 1st in the **ANALYZE 5.0** release.
B	width	int	Object map width. Must match volume width.
C	height	int	Object map height. Must match volume height.
D	depth	int	Object map depth. Must match volume depth.
E	nobjects	int	Number of defined objects. (1-256)
F1	name	char[32]	Object Name. (Null terminated string)
F2	display_flag	int	Display enabled(1) or disabled (0).
F3	copy_flag	u_char	Indicates if rotation, translation, or mirroring is applied to object(0) or copy of object(1).
F4	mirror	u_char	Indicates axis to mirror around. 1-xaxis, 2-yaxis, 4-zaxis. These are OR'd together to mirror on a combination of axes.
F5	status	u_char	Currently unused.
F6	n_used	u_char	Indicates if neighbors of different object types can be used in gradient calculations. (0-All neighbors used, 1-Only neighbors of same type used)
F7	shades	int	Number of lookup entries assigned to this object. A total of 250 is available for all objects.
F8	s_red	int	Currently unused. (Start Red)
F9	s_green	int	Currently unused. (Start Green)
F10	s_blue	int	Currently unused. (Start Blue)

TABLE 10-1

Field	Name	Type	Description
F11	e_red	int	Red component of object. (0-255)
F12	e_green	int	Green component of object. (0-255)
F13	e_blue	int	Blue component of object. (0-255)
F14	x_rot	int	Degrees of X rotation.
F15	y_rot	int	Degrees of Y rotation.
F16	z_rot	int	Degrees of Z rotation.
F17	x_shift	int	Voxels of X translation.
F18	y_shift	int	Voxels of Y translation.
F19	z_shift	int	Voxels of Y translation.
F20	x_center	int	Location of X center of rotation.
F21	y_center	int	Location of Y center of rotation.
F22	z_center	int	Location of Z center of rotation.
F23	i_xrot	int	X rotation increment.
F24	i_yrot	int	Y rotation increment.
F25	i_zrot	int	Z rotation increment.
F26	i_xshift	int	X translation increment.
F27	i_yshift	int	Y translation increment.
F28	i_zshift	int	Z translation increment.
F29	min_x	short int	Minimum X location.
F30	min_y	short int	Minimum Y location.
F31	min_z	short int	Minimum Z location.
F32	max_x	short int	Maximum X location.
F33	max_y	short int	Maximum Y location.
F34	max_z	short int	Maximum Z location.
F35	opacity	float	Object opacity. (.000001 - 1.)
F36	opacity_thick	int	Object thickness.
F37	dummy	int	Currently unused.

TABLE 10-1

Field	Name	Type	Description
			[Repeat all the F fields, once per object]
G	object_map	u_char	10.7

(The **`e_red`**, **`e_green`**, and **`e_blue`** values indicate the color assigned to the brightest pixel for this object. A linear ramp in computed, based on shades, down to a darker version of the same color. 10% of each RGB component is used as the computed s_red, s_green, and s_blue values.)

* Byte fields within int's and float's can be swapped on some machines. This includes all DEC machines and X86/Pentium processors.

10.5.1Note on Version Numbers

Several versions of the **ANALYZE** object map have been used with **ANALYZE** in previous releases. These versions are numbered and are indicated in the first header entry. The format described above is only the format for the latest release of **ANALYZE**, Version 5.0. The following describes the differences from previous releases:

FIGURE 10-5 **Object Map Version Numbers**

```
VERSION1 = 880102
No version number. Any # less than 880102, VERSION1 assumed.

VERSION2 = 880801
Version #'s added

VERSION3 = 890102
Run length compression added

VERSION4 = 900302
```

```
Start colors calculated rather than specified. Allowed specification of
color names rather than color values.Stored maximum and minimum bounds of
object. This allows quick movement of objects, because entire volume need
not be rendered.

VERSION5 = 910402
Switched from 8 possible objects (using one bit per object) to a possible
256 objects per object map.

VERSION6 = 910926
```

`Added` **`opacity`** `and` **`opacity_thick`** `parameters for true color transparency.`

10.6 Loading a Previously Defined Object Map

The **`AVW_LoadObjectMap()`** will load an object map previously saved to disk.

FIGURE 10-6 **Loading an Object Map**

```
AVW_ObjectMap *object_map;
char filepath[256];
     .
     .
     .
     object_map = AVW_LoadObjectMap(filepath);
     .
     .
     .
     AVW_DestroyObjectMap(object_map);
     object_map = NULL;
```

10.7 Creating New Objects

The **`AVW_AddObject()`** is used to create a new object in an object map. The **`NumberOfObjects`** is incremented and the new object (**`Object[NumberOfObjects-1]`**) is initialized. See Figure 10.7

10.8 Deleting Objects

An object within an object map can easily be deleted by calling the **`AVW_DeleteObject()`** function, specifying the object number to be deleted. When objects are deleted, all object numbers above the deleted object, are decremented and all voxels with in mapping volume which pointed to the deleted object are set to 255.

10.9 Extracting an Object from an Object Map

The **`AVW_GetObject()`** function can be used to extract an **`AVW_Volume`** from an object map. Any location within the object nap specified, which contains the specified object, will contain a one in the returned **`AVW_Volume`**. All the other location will contain zeros.

10.10 Using a AVW_Volume to Define an Object

The **`AVW_PutObject()`** does the opposite of **`AVW_GetObject()`**. Any non-zero location in the specified **`AVW_Volume`**, results in the corresponding location within the specified object map, being changed to the specified object. See Figure 10.7.

FIGURE 10-7 Using a Volume to Define a New Object

```
AVW_Volume *bin_volume;
AVW_ObjectMap *object_map;
char filepath[256];
     .
     .
     .
object_map = AVW_LoadObjectMap(filepath);
```

```
AVW_AddObject(object_map);
AVW_PutObject(bin_volume, object_map, object_map->NumberOfObjects-1);
AVW_SaveObjectMap(filepath, object_map);
AVW_DestroyObjectMap(object_map);
object_map = NULL;
```

10.11Using Ray-Casting to Define an Object

`AVW_RenderVolume()` can be used to define objects by setting the rendering **`Type`** to **`AVW_DELETE_VOXELS`** and specifying the appropriate **`DeleteDepth`** and **`DeleteValue`**. This is commonly used with a **`RenderMask`** to limit the area defined during the ray-casting process.

10.12Using Region Growing to Define an Object

`AVW_DefineConnected()` can be used by defining **`seed(s)`** and changing all enabled objects connected, which are within the specified threshold range, to the object specified.

FIGURE 10-8 **Using a region growing to define a new Object**

```
AVW_Volume *volume;
AVW_ObjectMap *object_map;
AVW_Point3 point;
AVW_PointList3 *seeds;
double tmin, tmax;
int count;
     .
     .
     .
seeds = AVW_CreatePointList3(1);
point.X = volume->Width/2;                      /* In a working program  */
point.Y = volume->Height/2;                    /* this would be selected */
point.Z = volume->Depth/2;                  /* interactively by the user */
AVW_AddPoint3(seeds, &point);
```

```
AVW_AddObject(object_map);
AVW_DefineConnected(volume, object_map, seeds, tmin, tmax,
                         object_map->NumberOfObjects-1, &count);
printf(“Number of Voxels Defined = %d\n”, count);
AVW_DestroyPointList3(seeds);
```

10.13 Manipulating the Mapping Volume with Other AVW Routines

Because the **`Volume`** which defines the mapping, is just like any other **`AVW_Volume`**, other volume manipulating routines in **`AVW`** can be used to read and/or write values in an object map.

FIGURE 10-9 Using AVW_Volume/AVW_Image Functions with Object Maps

```
int i, orient = AVW_TRANSVERSE;
AVW_Volume *volume;
AVW_ObjectMap *object_map;
AVW_Image *image = NULL;
     .
     .
     .
for(i=0;i<object_map->Volume->Depth;++i)
{
     image = AVW_GetOrthogonal(object_map->Volume, orient, i, image);
          .
          .  /* Process the slice */
          .
     AVW_PutOrthogonal(image, object_map->Volume, orient, i);
}
AVW_DestoryImage(image);
image = NULL;
```

CHAPTER 11

Multi-Spectral Classification

For two (or more) images in different modalities of the same subject in perfect spatial registration, the combined data set may be thought of as a single image in which the voxel values are vector quantities. Vector images in which the vector components (often called channels) are what we usually think of as spectral - such as red/green/blue color separation, or dual-energy X-ray CT - are often referred to as multi-spectral images. Images whose vector components reflect measurements of different physical quantities - such as correlated MRI/CT, or CT/SPECT, as well as images made of parameters derived from a single original - such as gradient/texture, may be also be analyzed as multi-spectral images.

Just as scalar images can be segmented by defining different extents of the gray scale to represent different tissue types, multi-spectral images can be segmented by defining different regions of the measurement space to be different tissue types or classes. The promise of multi-spectral analysis is that the dimensionally expanded measurement space will allow differentiations to be made which are impossible in any of the individual component images.

11.1 Classification Algorithms

The **AVW** Classification functions implement several common classification techniques including: *Gaussian Cluster, Neural Network*, *Nearest Neighbor, K-Nearest Neighbor* and *Parzen Windows*, which differ in complexity and basic approach. These algorithms offer a variety of trade-offs in accuracy, computation time and suitability to particular situations. Each algorithm assumes that certain voxels have been defined in the training sets as belonging to certain classes, and these voxels are used as samples of these classes. Unassigned voxels are then classified (or left unclassified) based on the estimated likelihood of their belonging to each of the defined classes. The algorithms are listed in rough order of increasing computation time, although this can vary significantly with the number of samples and for other reasons.

The *Gaussian Cluster* technique calculates means and standard deviations for each class along each feature, and then models the class' probability distribution as a gaussian function with these parameters. An unknown voxel is then classified by calculating the probability that it belongs to each class and choosing the largest result. The voxel is left unclassified if it lies farther than a certain number of standard deviations from all the class centers (controlled by the **MaxDist** parameter).

The *Neural Network* classifier builds a standard 3-layer feed forward neural network, of the kind commonly trained by backpropagation, and trains it on the class samples with a more advanced technique known as conjugate gradient optimization (see [1]).

The *Nearest Neighbors* technique simply takes each unassigned voxel, looks for its closest neighbor in feature space among the class samples, and assigns it to that same class. If two samples from different classes are equally close, the voxel is left unclassified. The *K-Nearest Neighbor* extends this idea to the k nearest neighbors among the class samples, giving each of these an equal vote. In both cases, the voxel is left unclassified if the winning samples are farther than the distance given by the **MaxDist** parameter.

The *Parzen Windows* technique is somewhat like *Gaussian Clustering* but makes a much more sophisticated estimate of each class' underlying probability distribution function from the samples. It uses these estimates to draw near-optimal decision boundaries, but is much slower than *Gaussian Clustering*. This latter technique, also known as a probabilistic neural network, is implemented in the manner described in [2].

The returned classified image or volume is always of data type **AVW_UNSIGNED_CHAR**. Its voxels are set to the class number to which that voxels was classified or to set 0. Voxels with value 0 were not classified. A histogram and voxel count of occurrences of each class for the classified image can be obtained as follows.

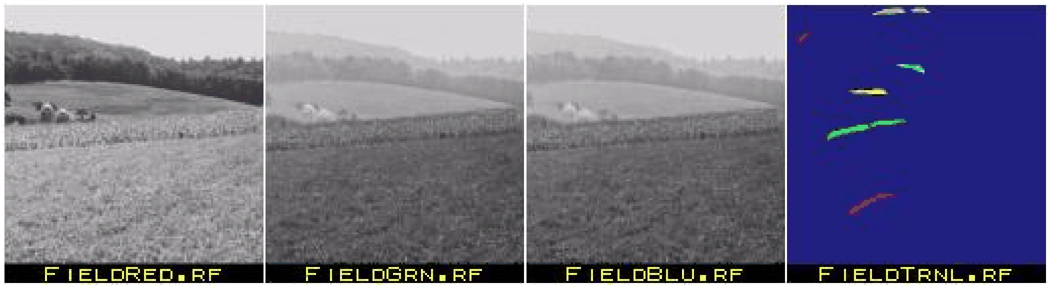
FIGURE 11-1 Histogram from a Classified Image

```
*AVW_Image *clsImg;
*AVW_Histogram *histo=AVW_NULL;
     .
     .                                    /* get a classified image clsImg */
     .
histo = AVW_GetImageHistogram(clsImg, AVW_NULL, 0, histo);
```

11.2 Input Requirements

Two or more spatially correlated images or volumes and a corresponding mask image or volume in which voxels of known classes are marked with the class value, are required to use the **AVW** multi-spectral classification functions. The following plate shows three bands from an RGB image, known regions are marked in the fourth image.

FIGURE 11-2 **Three Bands from an RGB Image with Training Image**



11.3 Classify Functions

`AVW_ClassifyVolume()` and **`AVW_ClassifyImage()`** classify multi-spectral volume or image data sets. Each function requires as input an array of pointers to volumes or images, the number of spectra, a training volume (or image), a specification of the classification type and some control parameters specific to each algorithm.

FIGURE 11-3 **Classifying an Image**

```
AVW_Image **image_list, *trnImg, *classified= NULL;
AVW_Image *image1, *image2, *image3;
int autotype = AVW_GAUSSIAN_CLUSTER, nimages = 3;
double MaxDist= 5.0;
double Sigma = 0.0, KValue = 0.0;                        /* Not used for */
int  Epochs = 0, HiddenEpochs=0;                 /* AVW_GAUSSIAN_CLUSTER */
     .
     .                                  /* load image1, image2, image3 */
     .
image_list = (AVW_Image **) malloc(nimages * sizeof(AVW_Image *));
image_list[0] = image1;
image_list[1] = image2;
image_list[2] = image3;
     .
     .                                                  /* get trnImg */
     .
classified = AVW_ClassifyImage(image_list, nimages, trnImg, autotype,
                 MaxDist, Sigma, KValue, Epochs, HiddenEpochs, classified);
```

The returned **`AVW_Image classified`** is returned as a classified image from the three images pointed at by **`**image_list.`** Classification is by the **`AVW_GAUSSIAN_CLUSTER`** algorithm. The **`MaxDist`** argument of 5.0 specifies the standard deviation of the gaussian window which is used to smooth the input samples to estimate the probability distribution. The other control parameters: **`Sigma`**, **`Kvalue`**, **`Epochs`**, and **`HiddenEpochs`** are ignored for **`AVW_GAUSSIAN_CLUSTER`** classification. The **`trnImg`** is a mask image used to indicate assigned classes for known pixels in the image. This image must be prepared, usually with user defined regions, prior to calling **`AVW_ClassifyImage()`**.

`AVW_ClassifyVolume()` is used in exactly the same manner for volumes.

FIGURE 11-4 **Classifying a volume**

```
AVW_Volume **vol_list, *trVol, *classified = NULL;
AVW_Volume *vol1, *vol2;
int autotype = AVW_NEURAL_NETWORK, nvols = 2;
int  Epochs = 200, HiddenEpochs=4;
double MaxDist= 0.0, Sigma = 0.0, KValue = 0.0;   /* Not used for NEURAL */
     .
     .                                          /* load vol1 and vol2 */
     .                                                  /* get trnVol */
vol_list = (AVW_Volume **) malloc(nvols * sizeof(AVW_Volume *));
vol_list[0] = vol1;
vol_list[1] = vol2;

classified = AVW_ClassifyVolume(vol_list, nvols, trVol, autotype, MaxDist,
                          Sigma, KValue, Epochs, HiddenEpochs, classified);
```

The returned **`AVW_Volume classified`** is returned as a classified volume from the two volumes pointed at by **`**vol_list.`** Classification is done using the **`AVW_NEURAL_NETWORK`** algorithm. The **`Epochs`** argument of **`200`** specifies the maximum number of passes through the class samples while training the neural network. **`HiddenEpochs`** of 4 specifies the number of hidden units in the (single hidden layer) neural network. The values of the other control arguments: **`MaxDist`**, **`Sigma`**, and **`KValue`** are ignored. The **`trnVol`** is a mask volume use to indicate assigned classes for known voxels in the volume. This volume must be prepared prior to calling **`AVW_ClassifyImage().`**

11.4 Training Images and Volumes

Training images and volumes are images and volumes in which voxels known to be in a class are assigned that class number and all other voxels are 0. Voxels initialized at 0 are identified by the classification functions as voxels to be classified. Preparation of training images and volumes is interface dependant and therefore not accomplished through **`AVW`** functions alone. In general preparing a training image involves creating an **`AVW_Image`** the same dimensions as the input image, setting all voxels to 0, and then setting voxels of known class to appropriate values.

FIGURE 11-5 **Preparing a Training Image**

```
AVW_Image *tr, *i1, *i2;
AVW_Point2 point;
int class_number;
     .
     .
     .
tr = AVW_CreateImage(NULL,i1->Width, i1->Height, AVW_UNSIGNED_CHAR);
AVW_SetImage(tr, 0.0);
     .
     .                      /* And for every known point in the training */
     .                  /* sample get point coordinates and class_number */

AVW_PutPixel(tr, &point, (double)class_number);
```

Class values should be consecutive and contiguous, i.e. if there are 4 classes, they should be numbered and have values of 1, 2, 3 and 4. All other pixels should be set to 0, to indicate that these are the pixels to be classified.

11.5 Control Parameters

AVW_ClassifyVolume() has the following parameters. Not all of them are used for each classification method. The parameters for **AVW_ClassifyImage()** are similar.

FIGURE 11-6 **Classify Volume Control Parameters**

```
AVW_Volume *AVW_ClassifyVolume(vols, numvols, train_vol, autotype,
                                            maxdist, sigma, kvalue, epochs,
                                                 hiddenepochs,  out_volume)
AVW_Volume **vols;                                 /* list of input volumes */
int numvols;                                     /* number of input volumes */
AVW_Volume *train_vol;                            /* masked training volume */
int autotype;                                   /* classification algorithm */
double MaxDist;
double Sigma;
double KValue;
int Epochs;
int HiddenEpochs;
AVW_Volume *out_volume;                                 /* recycled volume */
```

The **autotype** parameter specifies the classification algorithm and may be one of the following values: **AVW_GAUSSIAN_CLUSTER**, **AVW_NEURAL_NETWORK**, **AVW_NEAREST_NEIGHBOR**, **AVW_K_NEAREST_NEIGHBOR** or **AVW_PARZEN_WINDOWS**.

MaxDist specifies the maximum distance in feature space that an unknown point can be from a sample of a defined class to be considered as a possible member of that class. This argument is in units of standard deviations and is considered when **autotype** is **AVW_NEAREST_NEIGHBOR**, **AVW_K_NEAREST_NEIGHBOR** or **AVW_GAUSSIAN_CLUSTER**. It is ignored when **autotype** is **AVW_NEURAL_NETWORK** or **AVW_PARZEN_WINDOWS**.

Sigma specifies the standard deviation of the gaussian window used to smooth input samples to estimate the probability distribution when **autotype** is **AVW_PARZEN_WINDOW**. The value of **Sigma** is ignored for all other classification types. A default value of 5.0 is suggested.

KValue specifies the number of nearest neighbors considered when **autotype** is **AVW_K_NEAREST_NEIGHBOR**. The passed value of **KValue** is ignored for all other classification types.

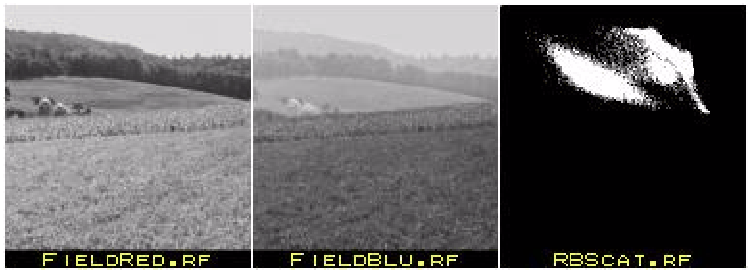
`Epochs` specifies the maximum number of passes through the class samples while training the neural network when autotype is **`AVW_NEURAL_NETWORK`**. The passed value of **`Epochs`** is ignored for all other classification types.

`HiddenEpochs` specifies the number of hidden units in the (single hidden layer) neural network when `auto-type` is **`AVW_NEURAL_NETWORK`**. The passed value of **`HiddenEpochs`** is ignored for all other classification types.

11.6 Scattergrams

Scattergrams are graphs of the measurement space for two spatially correlated images. Clusters in the feature space of a scattergram are visual clues for class separations. These images show in a single 2D image clusters of pairs of values. Pixel values from the two input images correspond to X and Y coordinates in the scattergram image. The X axis of the scattergram represents the gray level intensities of one image and the Y axis represents the intensities of the other image. Values in the scattergram are the number of occurrences of that value pair in the two input images. Scattergrams may require thresholding to view. Once voxel values in the scattergram image are changed to be class numbers; the processed scattergram can be used to very rapidly classify images.

FIGURE 11-7 **Scattergram Example**



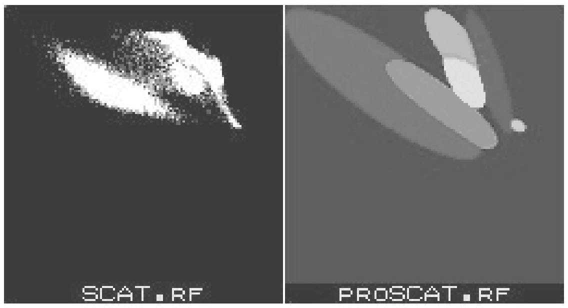
Scattergrams can be generated with **`AVW_Scattergram()`** if both input images are of the same size and have a data type of **`AVW_UNSIGNED_CHAR`**.

FIGURE 11-8 **Making a Scattergram**

```
AVW_Image *img1, *img2, *scat = NULL;
     .
     .                                                 /* load img1 and img2 */
     .
scat = AVW_Scattergram(img1, img2, scat);
```

This scattergram shows clustering of values corresponding to parts of the images. Once pixel values in the scattergram are set to class numbers, the processed scattergram can be used to classify the original images.

FIGURE 11-9 **A Processed Scattergram**

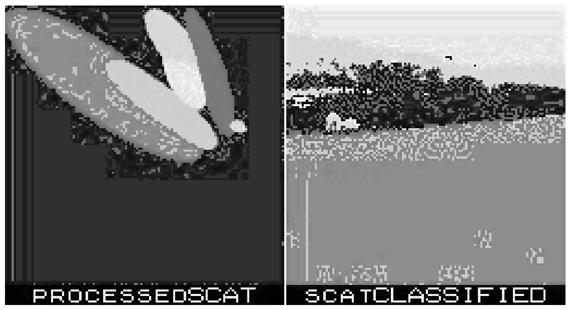


`AVW_ClassifyFromScattergram()` can be used to classify images:

FIGURE 11-10 **Classifying Images with a Processed Scattergram**

```
AVW_Image *img1=AVW_NULL, *img2=AVW_NULL, *scat=AVW_NULL *cls=AVW_NULL;
     .
     .              /* load img1 and img2 and processed scattergram: scat */
     .
cls = AVW_ClassifyFromScattergram(img1, img2, scat, cls);
```

FIGURE 11-11 **Classified Image**



11.7 Sample Programs

These sample programs with accompanying test images may be found in **`$AVW/extras`**:

classifyi.c	classifies an image from four input bands and one training image.
classifyRGB.c	separates an RGB image into three bands and then classifies
classifyv.c	classifies volumes from a multi-volume data set
scat.c	produces a scattergram from two input images
scatclass.c	classifies from two input images and a processed scattergram

References

A. Kramer and A. Sangiovanni-Vincentelli, “Efficient Parallel Learning Algorithms for Neural Networks”, in *Advanced in Neural Information Processing Systems 1*, pp 40-48, Morgan-Kaufmann, San Mateo, CA 1989.

D. F. Specht, “Probabilistic Neural Networks”, *Neural Networks*, Vol 3, pp 109-118, 1990.

M. Morrison and Y. Attikiouzel, "A Probabilistic Neural Network Based Image Segmentation Network for Magnetic Resonance Images", *Proc. Intl. Joint Conference on Neural Networks*, Baltimore, MD, June 7-11, 1992, pp III-60 - III-65, IEEE Press, 1992.

CHAPTER 12

Surface Matching

The full scientific, medical and educational value of multidimensional, multimodality imaging remains largely unexplored, and has been impeded by inadequate capabilities for accurate and reproducible registration and segmentation of 2D and 3D images. With regard to registration, 2D tomographic images taken at different times cannot be guaranteed to represent the same spatial section of the patient, and the 2D images themselves do not contain the information required to either measure or correct the misregistration. The use of 3D images makes full 6-degrees-of-freedom registration possible not only for images taken at different times, but for images from different modalities.

`AVW` provides registration capabilities through a single subroutine call to **`AVW_MatchSurfaces()`**. This routine employs a straightforward measure of registration cost applicable to arbitrarily complex multidimensional surfaces, and uses a multiscale/multiple starting point search strategy to significantly increase the probability of efficiently locating the best match without manual intervention (See References 1 and 2). Using a Chamfer matching method, two volumes are registered - one volume as the "base volume" and the other as the "match volume". The match volume will be subsequently transformed to align with the base volume and the resulting transformation matrix will be returned. Object contours are extracted from both the base and match volumes. If the base volume is non-isotropic, shape based surface interpolation is performed. A limited number of points are uniformly sampled from the match contour data set, and only these points are used for registration. Parameters are passed to the function using the **`AVW_MatchParameters`** structure and results are returned in the **`AVW_MatchResults`** structure.

12.1 AVW_MatchParameters Structure

The **`AVW_MatchParameters`** structure controls the initial position of the match surface and the range of translational and rotational motion allowed during the search for the best match. This structure is also returned as part of the **`AVW_MatchResult`** structure to allow for iterative matching.

FIGURE 12-1 **AVW_MatchParameters Structure From AVW_SurfaceMatch.h**

```
typedef struct
{
    int SamplePoints;
    int Centroid;
    float TranslationX, TranslationY, TranslationZ;
    float TranslationRange;
    float RotationPrecession, RotationNutation, RotationSpin;
    float RotationRange;
    float RotationInterval;
} AVW_MatchParameters;
```

`SamplePoints` determines how many randomly selected points on the match surface are used for the search. Larger numbers could result in better matches, but require more search time.

`Centroid`, if nonzero, indicates that the **`TranslationX`**, **`TranslationY`** and **`TranslationZ`** values are to be ignored, and the centroids of the base and match surface used for the initial search position.

`TranslationX`, **`TranslationY`** and **`TranslationZ`** specify the starting match position if **`Centroid`** is zero.

`TranslationRange` specifies the maximum translation (in voxels) allowed in the search.

`RotationPrecession`, **`RotationNutation`** and **`RotationSpin`** indicate the initial 3D rotational search position.

`RotationRange` specifies the total allowable range of rotation (about all three axes) for the search.

`RotationInterval` specifies the number of degrees per step in the final (high-resolution) search for the best matching position. The default is one degree.

12.2 AVW_MatchResult Structure

The **`AVW_MatchResult`** structure provides the specification of the best matching position found as well as a modified **`AVW_MatchParameters`** structure set up for a more rigorous (i.e. finer resolution) search over a restricted search range.

FIGURE 12-2 **AVW_MatchResult Structure from AVW_SurfaceMatch.h**

```
typedef struct
{
    AVW_Matrix Matrix;
    float MeanSquareDistance;
    AVW_MatchParameters *NextInput;
} AVW_MatchResult;
```

`Matrix` contains the entire translational, rotational and scaling transformation required to take the match image into the coordinate space of the base image.

`MeanSquareDistance` is the residual error of the current match. Since this absolute number is dependent upon the specific surfaces used for matching, it is only useful as a comparison of different matches of the same surfaces.

`NextInput` is an **`AVW_MatchParameters`** structure set up for a more rigorous search over a restricted search range.

12.3 Calling AVW_MatchSurfaces()

Figure 12-3 shows a complete example of a program which reads two volumes and then uses **`AVW_MatchSurfaces()`** to find the matrix necessary to register the match volume with the base volume. The inverse (See **`AVW_InvertMatrix()`**) of this matrix can be used to register the base volume to the match volume.

This example, uses hard coded file names for the base and match volumes. The **`AVW_MatchSurfaces()`** function supports only the **`AVW_UNSIGNED_CHAR`** data type and treats all non-zero voxels as the objects to match. This example prints the results to stdout, but the matrix and the original grayscale version of the match volume (or inverted matrix and base volume) could be passed the **`AVW_TransformVolume()`** function to obtain the registered volume.

FIGURE 12-3 **AVW_MatchSurfaces() Example**

```
/*
```

```
 * cc match.c -o match -I$AVW/include -L$AVW/$TARGET/lib -lAVW -lm
 */

#include <stdio.h>
#include <math.h>

#include "AVW.h"
#include "AVW_ImageFile.h"
#include "AVW_SurfaceMatch.h"

main()
{
char base_file[256];
char match_file[256];

AVW_Volume *base_volume;
AVW_Volume *match_volume;
AVW_MatchParameters mparams;
AVW_MatchResult *mresults;

AVW_ImageFile *db;

sprintf(base_file, "/Your/Path/To/The/BaseFile");
sprintf(base_file, "/Your/Path/To/The/MatchFile");

if((db = AVW_OpenImageFile(base_file, "r")) == NULL)
{
    AVW_Error("AVW_Open(base_file)");
    exit(1);
}

if(!(base_volume = AVW_ReadVolume(db, 0, NULL)))
{
    AVW_Error("ReadVolume");
    exit(1);
```

```
}

AVW_CloseImageFile(db);

if((db = AVW_OpenImageFile(match_file, "r")) == NULL)
{
    AVW_Error("AVW_Open(match_file)");
    exit(1);
}

if(!(match_volume = AVW_ReadVolume(db, 0, NULL)))
{
    AVW_Error("ReadVolume");
    exit(1);
}

AVW_CloseImageFile(db);

mparams.SamplePoints = 100;
mparams.Centroid = AVW_TRUE;
mparams.TranslationX = 0.0;
mparams.TranslationY = 0.0;
mparams.TranslationZ = 0.0;
mparams.TranslationRange = 100.0;
mparams.RotationPrecession = 0.0;
mparams.RotationNutation = 0.0;
mparams.RotationSpin = 0.0;
mparams.RotationRange = 100.0;
mparams.RotationInterval = 30.0;

if(!(mresults = AVW_MatchSurfaces(base_volume, match_volume, 1, &mparams)))
{
    base_volume = NULL;
    match_volume = NULL;
    AVW_Error("AVW_MatchSurfaces()");
    exit(1);
}

base_volume = NULL;                    /* Destroy if free_flag was false */
match_volume = NULL;                              /* Set to NULL if true */

printf("\nMeanSquareDistance = %f\n\n", mresults->MeanSquareDistance);

printf("Matrix = %7.3f %7.3f %7.3f %7.3f\n",
            mresults->Matrix[0][0], mresults->Matrix[0][1],
            mresults->Matrix[0][2], mresults->Matrix[0][3]);
printf("         %7.3f %7.3f %7.3f %7.3f\n",
            mresults->Matrix[1][0], mresults->Matrix[1][1],
            mresults->Matrix[1][2], mresults->Matrix[1][3]);
printf("         %7.3f %7.3f %7.3f %7.3f\n",
            mresults->Matrix[2][0], mresults->Matrix[2][1],
            mresults->Matrix[2][2], mresults->Matrix[2][3]);
printf("         %7.3f %7.3f %7.3f %7.3f\n",
            mresults->Matrix[3][0], mresults->Matrix[3][1],
            mresults->Matrix[3][2], mresults->Matrix[3][3]);

printf("\nIterate Parameters\n");
printf("SamplePoints = %d\n", mresults->NextInput->SamplePoints);
printf("Centroid = %d\n", mresults->NextInput->Centroid);
printf("X Translation = %7.2f\n", mresults->NextInput->TranslationX);
printf("Y Translation = %7.2f\n", mresults->NextInput->TranslationY);
printf("Z Translation = %7.2f\n", mresults->NextInput->TranslationZ);
printf("Translation Range= %7.2f\n",mresults->NextInput->TranslationRange);
printf("Precession = %7.2f\n", mresults->NextInput->RotationPrecession);
printf("  Nutation = %7.2f\n", mresults->NextInput->RotationNutation);
printf("      Spin = %7.2f\n", mresults->NextInput->RotationSpin);
printf("Rotation Range = %7.2f\n", mresults->NextInput->RotationRange);
printf("Rotation Interval= %7.2f\n",mresults->NextInput->RotationInterval);

free(mresults->NextInput);
free(mresults);
}
```

12.4 Sample Code

The above sample program is included in the **`$AVW/extras`**.

References

1. Jiang, H., K. Holton and R. Robb: Image registration of multimodality 3-D medical images by chamfer matching. Proceedings of Vision and Visualization, SPIE, San Jose, CA, Feb. 1992, pp. 649-659.

2. Robb, R., A.: Three-Dimensional Biomedical Imaging: Principles and Practice, VCH Publishers, Inc. New York, New York, 1995, pp. 188-203.

CHAPTER 13 Modeling

Currently, most real time simulations & high speed rendering engines require geometric shapes as their input. To be able to use 3D biomedical volumes in these simulations, we must first describe the objects of interest as sets of geometric primitives through the use of tiling algorithms. **AVW** provides methods to describe the surface of an object(s) in terms of triangles, quadrilaterals or as stacks of contours. These geometric surfaces accurately reflect the size, shape and position of the modeled object while containing a sufficiently small number of geometric entities (polygons) so as to permit real time manipulation of the surfaces on modern workstations.

13.1 The Algorithms

AVW supports 3 tiling algorithms, one based on Kohonen Networks, one based on Competitive Hebbian Learning and Marching Cubes[1].

13.1.1 Kohonen Based Tiling

A Kohonen network (also known as a self organizing map [2]) is a common type of neural network which maps sample vectors **S** from an N-dimensional space of real numbers $\mathbf{R}^{\mathbf{N}}$ onto a M-dimensional array of nodes **L** (where $M \leq N$) in an ordered fashion. An N-dimensional position vector **N** is attached to every node in **L**. An arbitrary N-dimensional vector $v \in V$ is then mapped onto the node which lies nearest to it in $\mathbf{R}^{\mathbf{N}}$. This node is referred to as the best matching unit (BMU) and satisfies equation 1.

$$\|bmu - v\| \leq \|n - v\|, (\forall(n \in S)) \text{u} \tag{EQ 1}$$

This defines the mapping:

$$V \to L, (v \in V) \to (T(v) \in L) \tag{EQ 2}$$

where

$$T(v \in V) = bmu \qquad \text{(EQ 3)}$$

By applying **T** to a given set of sample vectors **S**, **V** is divided into regions with a common nearest position vector $n \in L$. This is known as Voroni Tessellation and the regions are denoted Vorini Regions.

The usefulness of the network is that the resultant mapping preserves the topology and distribution of the position evocators. That is, to preserve topology, adjacent vectors in $\mathbf{R^N}$ are mapped to adjacent nodes in **L** and adjacent nodes in **L** will have similar position vectors in $\mathbf{R^N}$. Now, let **P(x)** be an unknown probability distribution on $\mathbf{R^N}$ from which we draw any number of sample vectors. Then to preserve distribution, for any sample vector from **P(x)** each node has the same probability of being mapped to that sample vector. This means that the relative density of position vectors in $\mathbf{R^N}$ approximates **P(x)**.

The tiling process begins with the generation of an initial surface consisting of a predefined number of polygons in a satirical configuration encompassing the bounding box of the volumetric object. The 3D coordinates of the polygonal vertices are used as the initial position vectors **N**. The network is adapted to a feature set (a set of sample vectors **S**) derived from the volumetric data. By weighting the members of the feature set, the position vectors will adapt themselves to the areas of the feature set with the highest weight, thus capturing the “most important” details of the feature set to the extent possible within a limited polygonal budget.

13.1.2Growing Networks

This algorithm is adapted from the “growing cell structures” and “growing neural gas” algorithms first described by Fritzke[3,4]. It is based on the successive addition and deletion of nodes to/from an initially small 2D Kohonen Network by evaluating local statistical measures gathered during previous steps in the adaptation process.

The network is adapted to the set **S** of sample vectors by a Competitive Hebbian Learning model which determines how new edges (connections) are added to the network. The topological structure of the network is defined by a set of unweighted edges **E** among pairs of nodes. An aging scheme is used to remove obsolete edges during adaptation. A polygon **p** is defined by any set of 3 nodes which are connected together by a set of 3 edges. The polygonal surface is therefore, the set of all polygons **P**.

Tiling starts by placing 3 nodes **a**, **b** and **c** at random positions $\mathbf{v_a}$, $\mathbf{v_b}$ and $\mathbf{v_c}$ in $\mathbf{R^N}$. A sample input signal $s \in S$ is generated and the nearest node $\mathbf{n_1}$ and the second nearest node $\mathbf{n_2}$ are determined ($\mathbf{n_1}$ and $\mathbf{n_2}$ are referred to as the best matching units or **bmus**). Then the are of all edges emanating from $\mathbf{n_1}$ are incremented by:

$$\Delta err(n_1) = \|v_{n_1} - s\|^2 \tag{EQ 4}$$

n$_1$ and its direct topological neighbors (the direct topological neighbors of a given node **n** is **DTN(n)** which is defined to be the set of all nodes which are connected to **n** by a single edge) are moved towards **s** by fractions ε_b and ε_n, respectively, of the total distance.

$$\Delta v_{n_1} = \varepsilon_b(s - v_{n_1}) \tag{EQ 5}$$

$$\Delta v_n = \varepsilon_n(s - v_n) \quad \forall n \in DTN(n_1) \tag{EQ 6}$$

If **n$_1$** and **n$_2$** are connected by an edge, the connecting edge’s age is set to 0. If there is no connecting edge, one is created as well as all possible polygons resulting from the addition of this edge thus insuring a triangulated network and preventing the accidental creation of holes. Any edge with an age exceeding **a$_{max}$** is removed as are the polygons formed by that edge and any nodes that either have no emanating edges or edges that belong to zero polygons.

If the number of input signals generated to this point is an integer multiple of λ, the node addition frequency, a new node **r** is inserted between the nodes **q** and **q**’s direct neighbor **f**, **q** and **f** are the nodes where **v$_r$** is maximized (see equation 7).

$$v_r = \alpha(v_q + v_f) \tag{EQ 7}$$

Edges are inserted to connect **r** with **q** and **f** (the original edge between **q** and **f** is deleted). If needed, additional edges and polygons are added to the network in order to maintain the network as a connected set of simplices. The error variables **v$_q$** and **v$_f$** are decremented by multiplying them by the constant α. **v$_r$** is set to **v$_q$**. At this point, all error variables are decreased by multiplying them by the constant **d**[1] and if the polygonal budget has not been reached the adaptation process is repeated.

13.1.3Post-Processing

There is little post-processing done on surfaces generated using the Kohonen method. Since the network is held in a fixed topology, it is possible to define the surface so that all surface normals are pointing in the same direction. However, this is not the case for the Growing Net method.

1. The constants d and α are empirically found to be 0.995 and 0.5 respectively.

The Growing Net algorithm requires a post-processing step to re-organize the polygonal surface so that all the surface normals are pointing in the same direction. To do this, the polygonal surface is transformed into a weighted connected graph where each node in the graph represents a polygon and adjacent polygons are connected by a graph edge. The graph is rooted by the polygon with the largest Z coordinate and a weight $\mathbf{W_{ij}}$ is assigned to the graph edge connecting the i^{th} and j^{th} polygons and is given by:.

$$W_{ij} = 1 - \left|\vec{n}_i \cdot \vec{n}_j\right| \quad \text{(EQ 8)}$$

Where $\vec{n}_i$ and $\vec{n}_j$ are the surface normals for their respective polygons. Small weights are indicative of surface normals that are nearly parallel, thus configuring the graph as a minimal spanning tree will tend to propagate changes in orientation along directions of low curvature avoiding the ambiguous situations that may result when trying to propagate changes in orientation along sharp edges.

To re-orient the surface normals, the root polygon's surface normal is set to point towards the positive Z axis (outside) and then the graph is traversed in a depth first order. Each subsequent polygon's surface normal is changed to be consistent with that of its parent. That is, if the current polygon **i** has normal $\vec{n}_i$ and its child, polygon **j** has normal $\vec{n}_j$, $\vec{n}_j$ is set to $-\vec{n}_j$ if $\vec{n}_i \cdot \vec{n}_j < 0$.

13.2 Tiling Parameters

AVW provides a single interface, **AVW_TileVolume()**, to the tiling algorithms. The tiling process performed by **AVW_TileVolume()** occurs in three phases: surface detection, feature extraction and polygonization. Algorithm selection and parameter definition is controlled through the use of the **AVW_TileParameters** structure.

FIGURE 13-1 **AVW_TileParameters Structure**

```
typedef struct
    {
    AVW_Volume *Vol;
    AVW_ObjectMap *Omap;
    unsigned int Type;
    unsigned int AddNodeFreq;
    unsigned int MaximumAge;
    unsigned int KohonenShapeOrient
    unsigned int KohonenFlag;
    unsigned int PolygonBudget;
    unsigned int KohonenRepetitions;
    unsigned int KohonenNeighborhood;
    int Mask;
    int CurveOpRadius;
    int CloseSrfcFlag;
    int KohonenMajorAxis;
    float KohonenShapeOffset;
    float KohonenNeighborRadius;
    float KohonenAlpha;
    float Eb, En;
    float GrowingAlpha;
    float GrowingD
    } AVW_TileParameters;
```

AVW_InitializeTileParameters() provides a means of initializing all the tiling parameters to reasonable values, thus the user need only alter those parameters specific to his task.

FIGURE 13-2 **Initialization Example**

```
AVW_Volume *volume = NULL;
AVW_ObjectMap *objectMap = NULL;
AVW_TileParameters *tp = NULL;
.
.
.
tp = AVW_InitializeTileParameters(volume,objectMap,NULL);
if (tp == NULL)
{
AVW_Error(“AVW_InitializeTileParameters”);
break;
}
.
.
```

.

13.2.1Tile Type

The **Type** parameter specifies which tiling algorithm to use. The available options are shown in the figure below and the following table. By default, this parameter will be set to **AVW_TILE_GROW**.

FIGURE 13-3 Tile Types Defined in AVW_Model.h

```
#define AVW_TILE_KOHONEN         0
#define AVW_TILE_GROW            1
#define AVW_MARCHING_CUBES       2
```

TABLE 13-1 Tile Type Algorithms

Type	Description
AVW_TILE_KOHONEN	The tiled surface will be created using a Kohonen Net to map a mesh to the object's surface. The network adapts a fixed topology to the surface of the object. While this preserves the probability density of the inflection points of the object's surface, the network is unable to correctly tile surfaces containing holes or bifurcations.
AVW_TILE_GROW	The tiled surface will be created using a polygon growing algorithm through a Competitive Hebbian Learning process. This algorithm is similar to the Kohonen Net in that it preserves the probability density of the surface's inflection points. However instead of trying to adapt a fixed topology to the object's surface, the algorithm attempts of grow a surface through the addition/deletion of nodes to a triangulated connected network. This allow the algorithm to correctly tile surfaces contained holes and bifurcations.
AVW_MARCHING_CUBES	The tiled surface is created using a standard marching cubes algorithm. This is an implementation of the algorithm described by Lorensen & Cline.

13.2.2Tile Mask

Surface detection is based on a simple thresholding operation. The object of interest is considered to be those voxels whose value is greater than or equal to some masking value, and the object's surface is the set of voxels where the change between background (those voxels whose value is less than the masking value) and the object occurs. The **Mask** parameter is used to specify the masking value of the object within the volume to be tiled. If the **ObjectMap** parameter of **AVW_InitializeTileParameters()** is null then the **Mask** parameter used as a thresholding value otherwise, the parameter represents an object mask within the object map.

13.2.3Tile CurveOpRadius

For both the **AVW_TILE_KOHONEN** and **AVW_TILE_GROW** algorithms, a feature extraction step is required prior to polygonization. This step calculates local surface curvature measure for each voxel on the object's surface and eliminates those voxels that are locally flat.

FIGURE 13-4 Calculation of Local Curvature

```
Given a binary volume f, then the curvature c is calculated by applying the
following equation to all voxels fxyz in the volume.
```

$$c = \sum_{z=-1}^{1} \sum_{y=-1}^{1} \sum_{x=-1}^{1} f_{xyz} \qquad \text{(EQ 9)}$$

These surface curvature (inflection) points define the set of sample vectors $s \in S$ which represent a 2D manifold embedded in $\mathbf{R}^3$. This 2D manifold represents the surface to be tiled.

The **CurveOpRadius** parameter determines the spacing between the elements of the curvature operator. The default value of 1 sets the operator to act as a 26 connected nearest neighbor kernel. To reduce the effects of local surface noise, the spacing between the 26 elements of the operator may be increased, however increasing the spacing between the elements will also reduce the operator's sensitivity to rapid changes in surface curvature.

13.2.4Tile CloseSrfcFlag

If the **CloseSrfcFlag** is set to **AVW_TRUE** then the resultant surface will be a closed rather than open ended surface. Then default is to produce an open ended surface.

13.2.5Tile KohonenMajorAxis

The **KohonenMajorAxis** parameter specifies which axis is to be considered the data's major axis (axis of greatest extent). It may be set to ten of the options given in the figure below. The default is **AVW_ZAXIS.**

FIGURE 13-5 Major Axis Types Defined in AVW.h

```
#define AVW_XAXIS                  1
```

```
#define AVW_YAXIS                2
#define AVW_ZAXIS                4
```

13.2.6Tile KohonenShapeOrient

The **KohonenShapeOrient** parameter specifies the initial orientation of the network. This parameter may be set to one of the values given in the following figure. A transverse orientation is the default. This means that the network will be oriented so that the majority of the adaptation (all of the adoption is the network is prevented from training along the data's major axis) will take place in the transverse plane.

FIGURE 13-6 **Shape Orientation Types Defined in AVW.h**

```
#define AVW_TRANSVERSE           0
#define AVW_CORONAL              1
#define AVW_SAGITTAL             2
```

13.2.7Tile KohonenShapeOffset

KohonenShapeOffset specifies a multiplier used in determining the initial distance between the object's surface and the network. If **KohonenShapeOffset** is set to a value less than 1 the network will be placed within the object's surface (adaptation will then cause the network to expand out to the surface) and a value greater than 1 will place the network outside of the surface. The actual distance is determined by multiplying the radii of the 2-D bounding oval (determined on a slice by slice basis along the orientation specified by **KohonenShapeOrient**) of the object by **KohonenShapeOffset**.

13.2.8Tile KohonenFlag

The parameter **KohonenFlag** is set by oring together the flags given in the following figure and table. This parameter is used to fine tune the adaptation process.

FIGURE 13-7 **Kohonen Network Flags Defined in AVW_Model.h**

```
#define AVW_TRAIN_IN_MAJOR_AXIS  1
#define AVW_TRAIN_WITH_WEIGHT    2
```

TABLE 13-2 **Kohonen Network Flags**

Type	**Description**
AVW_TRAIN_IN_MAJOR_AXIS	This flag will cause the network to adapt along the object's major axis. Setting this flag allows for some additional refinement of the final geometric surface, however by allowing the network to adapt in 3 dimensions there is the risk that the network will collapse into the interior of the object reducing the volume contained within the geometric surface and reducing the accuracy of the resultant model.
AVW_TRAIN_WITH_WEIGHT	By default, the network will adapt to the surface's inflection points in an unbiased fashion. If this flag is set, the network will use the magnitude of the inflection points as a weighting factor and will favor points with the highest curvature during adaptation.

13.2.9Tile PolygonBudget

The parameter **PolygonBudget** is used by the **AVW_TILE_KOHONEN** and the **AVW_TILE_GROW** algorithms in determining the maximum number of polygons that may be used in the production of a geometric surface. Unless **CloseSrfcFlag** is set, the resultant surface will contain no more than **PolygonBudget** polygons. Setting **CloseSrfcFlag** will add the capping polygons to the total number of polygons in the surface.

13.2.10 Tile KohonenRepetitions

The parameter **KohonenRepetitions** specifies the number of times the data is presented to the network during the adaptation process. In theory, the more times the data is presented to the network, the more accurate the final product will be. However, in practice there is a point beyond which the computational cost of adaptation out weighs the amount of refinement performed by that step. While this point is dependent on a number of factors, in general, adequate results may be obtained by setting **KohonenRepetitions** to a value within the range of 4 - 8 (by default this parameter is set to 4).

13.2.11 Tile KohonenNeighborhood

The **KohonenNeighborhood** parameter controls how a "neighborhood" of cells within the network will move during adaptation. The possible values **KohonenNeighborhood** may be set to are given in the figure and table presented below.

FIGURE 13-8 **Neighborhoods Defined in AVW_Model.h**

```
#define AVW_BUBBLE_NEIGHBORHOOD             0
#define AVW_GAUSS_NEIGHBORHOOD              1
#define AVW_TRIANGLE_NEIGHBORHOOD           2
```

TABLE 13-3 Neighborhood Definitions

Type	Description
AVW_BUBBLE_NEIGHBORHOOD	Acts as a rectangular step function. During adaptation, the entire neighborhood will move the same distance toward the current inflection point.
AVW_GAUSS_NEIGHBORHOOD	The distance any member of the neighborhood moves is determined by a gaussian function based on its distance from the center of the neighborhood
AVW_TRIANGLE_NEIGHBORHOOD	The distance any member of the neighborhood moves is determined by a triangular step function based on its distance from the center of the neighborhood

13.2.12 Tile KohonenNeighborRadius

The **KohonenNeighborRadius** parameter specifies the initial radius of a network neighborhood. If this parameter is less than 1, the initial radius will be set to be one third of the distance around a 2D bounding oval orthogonal to the objects major axis by **AVW_TileSurface**(). By default, this parameter is set to 0.

13.2.13 Tile KohonenAlpha

The **KohonenAlpha** parameter specifies the initial learning rate during adaptation. The learning rate is used in calculating the distance each member of a network neighborhood will move during each adaptation step.

13.2.14 Tile AddNodeFreq

The **AddNodeFreq** parameter is used by the growing net algorithm to determine the number of adaptation steps that must occur before a new node is added to the system.

13.2.15 Tile MaximumAge

The **MaximumAge** parameter specifies the maximal age allowed for the growing net edges. Edges with an age greater than **MaximumAge** are removed from the system.

13.2.16 Tile Eb and En

Eb and **En** determine the adaptation factors for the best matching unit node (**Eb**) and its direct neighbors (**En**).

13.2.17 Tile GrowingAlpha and GrowingD

The parameters **GrowingAlpha** and **GrowingD** specify the factors for decreasing the error counters of the first and second best matching nodes (**GrowingAlpha**) and for decreasing the error counters of all the nodes(**GrowingD**) after the insertion of a new node into the system.

13.3 Tiled Surfaces

13.3.1AVW_TileSurface Structure

The **AVW_TiledSurface** structure describes a 3D polygonal surface. It consists of a list of 3-space coordinates contained in **Coords** and a list of interconnections given by **Indices**.

FIGURE 13-9 **AVW_TiledSurface Structure Defined in AVW_Model.h**

```
typedef struct
    {
    AVW_FPointList3 *Coords
    unsigned int NumberOfNodes
    unsigned int MaximumNodes
    unsigned int BlockSize
    int Indices
    } AVW_TiledSurface;
```

Thus, a surface consisting of a single unit quadrilateral polygon would contain (0,0,0) (0,1,0) (1,1,0) and (1,0,0) in **Coords** and [0,1,2,3, **SRFC_END_INDEX**] in **Indices**. The **NumberOfNodes** variable would be set to 4. The parameter **MaximumNodes** determines the maximum number of indices the structure can hold before automatic reallocation occurs. When reallocation occurs, it will happen in **BlockSize** increments.

13.4 Creation and Destruction of Tiled Surfaces

Tile surfaces are derived from volumetric data through the use of **AVW_TileVolume()**. This requires initializing a set of tiling parameters via **AVW_InitializeTileParamters()**. A tiled surface is destroyed by **AVW_DestroyTiledSurface()**. The following figure illustrates the creation and destruction of a tiled Surface.

FIGURE 13-10 **Creating and Deleting a Tiled Surface**

```
AVW_Volume *volume = NULL;
AVW_ObjectMap *omap = NULL;
AVW_TileParameters *tp = NULL;
AVW_TiledSurface *srfc = NULL;
.
.
.
tp = AVW_InitializeTileParameters (volume, omap, NULL);
if (tp == NULL)
{
AVW_Error(“AVW_InitializeTileParameters”);
/* handle error, perhaps exit? */
}
tp->PolygonBudget = 1500;
tp->Type = AVW_TILE_GROW;
tp->Mask = 50
.
.
.
srfc = AVW_TileVolume(tp, NULL);
if (srfc == NULL)
{
AVW_Error(“AVW_TileVolume”);
/* handle error, perhaps exit? */
}
.
.
```

```
.
AVW_DestroyTiledSurface (srfc);
```

13.5 Saving a Tiled Surface to Disk

AVW_SaveTiledSurface() is used to write a tiled surface to disk and supports several common CAD formats as well as SGI's Inventor and VRML. A list of formats is given in the following table. Unlike other AVW save functions, **AVW_SaveTiledSurface()** does not automatically provide a file extension in the event the file name provided to it lacks one.

FIGURE 13-11 **Saving a Tiled Surface**

```
AVW_TiledSurface *srfc = NULL;
char *outfile = “test.dxf”
int rslt
.
.
.
rslt = AVW_SaveTiledSurface(srfc, outfile, AVW_INVENTOR_SURFACE);
if (rslt != AVW_SUCCESS)
{
AVW_Error (“AVW_SaveTiledSurface”);
/* handle error here, perhaps exit? */
}
```

FIGURE 13-12 **Supported Output File Formats Defined in AVW_Model.h**

```
#define AVW_DXF_SURFACE                3
#define AVW_INVENTOR_SURFACE           1
#define AVW_OBJ_SURFACE                4
#define AVW_POLY_SURFACE               5
#define AVW_STL_SURFACE                6
#define AVW_VRIO_SURFACE               0
#define AVW_VRML_SURFACE               2
```

TABLE 13-4 **File Formats**

Type	Common File Extension	Description
AVW_DXF_SURFACE	.dxf	AutoCAD
		Supports a subset of ASCII Drawing Interchange Format (DXF) 12.[a]
AVW_INVENTOR_SURFACE	.iv	IRIS Inventor
		Supports a subset of ASCII format IRIS Inventor 2.0.[a]
AVW_OBJ_SURFACE	.obj	Alias Wavefront Model Format
		Supports a subset of the ASCII OBJ format[a]
AVW_POLY_SURFACE	.poly	SGI Poly
		A line-structured ASCII format for the description of geometric polyhedral data.
AVW_STL_SURFACE	.stl	3D Lithography
		A line-structured ASCII format used to define 3D solids for use in stereo lithographic systems.
AVW_VRIO_SURRFACE	.vrio	VRIO
		Developed in house as the “native” format for AVW_TiledSurface. It is a ASCII text file with a format similar to SGI Inventor
AVW_VRML_SURFACE	.wrl, .vrml	VRML
		ASCII format used for data transfer via HTML and the world wide web.

a. Properties beyond simple polygon description are not supported

13.6 *Loading Polygonal Data as a Tiled Surface*

AVW_LoadTiledSurface() will load a tiled surface from disk and supports several common CAD formats. Like **AVW_SaveTiledSurface()**, **AVW_LoadTiledSurface()** does not automatically append a file extension to a file name.

FIGURE 13-13 **Loading a Tiled Surface**

```
    AVW_TiledSurface *srfc = NULL;
    char *infile = “test.dxf”
.
.
.
    srfc = AVW_LoadTiledSurface(infile,AVW_DXF_SURFACE,NULL);
    if (srfc == NULL)
    {
    AVW_Error(“AVW_LoadTiledSurface”);
    /* handle error here, perhaps exit? */
    }
```

FIGURE 13-14 **Supported Input File Formats**

```
    #define AVW_DXF_SURFACE                3
    #define AVW_OBJ_SURFACE                4
    #define AVW_POLY_SURFACE               5
    #define AVW_STL_SURFACE                6
    #define AVW_VRML_SURFACE               2
```

13.7 Interface

Tiled surfaces may be interfaced with volumetric data through the use of **AVW_DrawTiledSurface()**. Given a tiled surface and a volume, **AVW_DrawTiledSurface()** will set all the voxels in the volume that fall along polygonal edges to a user defined value. This has the effect of placing a wiregrid representation of the surface into the volumetric domain.

13.8 Slice-based Surface Representations

For some applications, it is preferable to represent a surface as a set to stacked contours rather than a set of polygons. This $2\frac{1}{2}$ D representation is often used in rapid prototyping machines where solid material is placed between a set of inner and outer boundaries. AVW supports slice-based surface representations through the **AVW_RPParam** structure and **AVW_SliceVolume()**. This function will extract the set of contour boundaries that describe a given object from a volume and write those boundaries to a file in one of the following formats:

FIGURE 13-15 **Supported Contour File Formats Defined in AVW_Model.h**

```
#define AVW_HPGL_SURFACE              7
#define AVW_POGO_SURFACE              8
#define AVW_SLC_SURFACE               10
#define AVW_SSD_ASCII_SURFACE         11
```

TABLE 13-5 **Contour File Formats**

Type	Common File Extension	Description
AVW_HPGL_SURFACE	.hpp	A modified version of HPGL. This ASCII file format uses the ‘PG’ mnemonic to indicate the separation of layers. Contours within each layer are separated by the ‘PU’/’PD’ mnemonics.
AVW_POGO_SURFACE	.slc	A simplified version of the standard 3D Systems’ SLC. This binary file format lacks the header information of the standard SLC format.
AVW_SLC_SURFACE	.slc	The standard 3D Systems’ SLC format. This binary file format consists of layers of contours, with each layer consisting of a series of nested loops. Each loop represents a single boundary. The format includes an ASCII header that contains information on the part and how it was prepared. The format follows the IEEE floating point format and has Intel-endian alignment
AVW_SSD_ASCII_SURFACE	.txt	This is the ASCII version of the standard Analyze^{tm} SSD contour file format.

13.9 Contour Extraction

FIGURE 13-16 Extracting Contours

```
AVW_Volume *vol = NULL;
AVW_ImageFile *db = NULL;
AVW_RPParam *rp = NULL;
.
.
.
if ((rp = AVW_InitializeRPParam(vol,rp)) == NULL) {
/* handle error here, perhaps exit? */
}
rp->MaskValue = 100;
rp->Format = AVW_SLC_SURFACE;
rp->SubvolumeFlag = AVW_FALSE;
if (AVW_SliceVolume(rp, filename)<= AVW_SUCCESS) {
/* handle error here, perhaps exit? */
}
AVW_DestroyRPParam(rp);
```

In the figure above, **AVW_InitializeRPParam()** is used to initialize and create an **AVW_RPParam** structure. Once the necessary parameters have been modified, the structure is passed to **AVW_SliceVolume()** and the contours are written to a file in the desired format. **AVW_SliceVolume()** will append the appropriate extension to the file name if the filename is missing an extension. The **AVW_RPParam** structure is destroyed by **AVW_DestroyRPParam()**.

13.10 Contour Parameters

AVW provides a single interface to the contour extraction algorithms: **AVW_SliceVolume()**. The actual process of contour extraction is determined by the **AVW_RPParam** structure. **AVW_InitializeRPParam()** provides a means of initializing all the contour extraction parameters to reasonable values, thus the user need only alter those parameters specific to his task.

FIGURE 13-17 **The AVW_RPParam Structure**

```
typedef struct
     {
     AVW_Volume *MaskVolume;
     int         InterpolateFlag;
     int         SubvolumeFlag;
     int         Format;
     int         Orientation;
     double      MaskValue;
     double      AngleResolution;
     } AVW_RPParam;
```

13.10.1 Contours MaskValue

The **MaskValue** parameter specifies the mask value of the object whose contours are to be extracted.

13.10.2 Contours Format

The **Format** parameter specifies the format of the output file. This may be set to any of the formats listed in Figure 13-15 on page 148.

13.10.3 Contours Orientation

This parameter specifies the orientation of the slices during the contour extraction process. It may be set to any of the values given in Figure 13-6 on page 140.

13.10.4 Contours SubvolumeFlag

If set, **AVW_SliceVolume()** will subvolume the dataset so that the object of interest is contained within a minimum enclosing volume. This may cause the data to be reoriented prior to contour extraction. The origin for the contours will be the origin of the new subvolume and not the origin of the original volume. Subvol-uming is useful when defining parts that will eventually be built on a rapid prototyping machine. By default, this parameter is set to **AVW_TRUE**.

13.10.5 Contours InterpolateFlag

If set, **AVW_SliceVolume()** will use tri-linear interpolation rather than nearest neighbor interpolation when generating the subvolume. By default, this parameter is set to **AVW_TRUE**.

13.10.6 Contours AngleResolution

The **AngleResolution** parameter specifies the step size in degrees of rotation used to search for the minimum enclosing brick during the subvoluming process.

13.11References

1. Lorensen, W. E. and H. E. Cline: Marching Cubes: A High Resolution 3D Surface Construction Algorithm, Computer Graphics, 21(4), July 1987

2. Kohonen, T: Self-Organization and Associative Memory (3rd Ed), Springer-Verlag, Berlin, 1989

3. Fritzke, B.: Let it Grow - Self Organizing Feature Maps With Problem Dependent Cell Structure, Proc. of the ICANN-91, Helsinki, 1991

4. Fritzke, B.: A Growing Neural Gas Network Learns Topologies, in Advances in Neural Information Processing Systems 7, Eds. G. Teasauro, S Touretzky and T. K. Leen, MIT Press, Cambridge MA, 1995

13.12Appendix

FIGURE 13-18 **dumpslc.c: sample code to dump slc formatted files to ASCII**

```
/*
 * dumpslc
 * cc -o dumpslc dumpslc.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0
#define POINT_BUFFER_SIZE 4096
```

```
#define RESERVED_SIZE 256

#ifdef SWAP
#define DOSWAP TRUE
#else
#define DOSWAP FALSE
#endif

const max_vertices = 2048;

struct sample_table_entry {
    float min_z_level;
    float layer_thickness;
    float beam_comp;
    float reserved;
};

int readAndSwap (char *buffer, int size, int count, FILE *file,
                  unsigned int reverse)
{
    char *s;
    int halfsize;
    char temp;
    int i,j,k,sizeprime;
    int n;

    n = fread(buffer,size,count,file);
    if (!reverse || n < count || size <2) return n;
    halfsize = size >> 1;
    sizeprime = size-1;
    s = (char *)buffer;
    for (i = 0; i < count; i++,s+=size) {
        for (j = 0; i < halfsize; j++) {
            k = sizeprime - j;
            temp = s[j];
```

```
              s[j] = s[k];
              s[k] = temp;
          }
     }
     return n;
}

void read_slc_header (FILE *fp)
{
     char head_char;
     while ((head_char = getc(fp)) != 0x1A) putchar(head_char);
     printf("\n");
}

void read_reserved_section(FILE *fp)
{
     char key_char;
     int i;
     for (i = 0; i < RESERVED_SIZE; i++)
          key_char = getc(fp);
}

void read_table_entry(FILE *fp)
{
     int num_read, i;
     char table_entry_size;
     struct sample_table_entry sample_table;
     num_read = readAndSwap((char *)&(table_entry_size),
                    sizeof(table_entry_size),1,fp,DOSWAP);
     if (table_entry_size == 0) {
          printf ("sample table size is 0");
          exit(1);
     }
     printf ("Sample table size : %d\n\n",table_entry_size);
     printf ("Minimum Z level\tLayer Thickness\tLine Width\tReserved\n");
```

```
    for (i = 0; i < table_entry_size; i++) {
        num_read = readAndSwap((char *)&(sample_table),
                sizeof(float),4,fp,DOSWAP);
        printf (“%7.3f\t%7.4f\t%7.4f\t%6.3f\n”,
            sample_table.min_z_level,
            sample_table.layer_thickness,
            sample_table.beam_comp,
            sample_table.reserved);
    }
    printf (“\n”);
}

void print_points(float buffer[], unsigned int numPoints)
{
    int i;
    for (i = 0; i < (numPoints * 2); i+=2)
        printf (“%9.6f, %9.6f\n”, buffer[i], buffer[i+1]);
}


void process_points(long unsigned int numPoints, FILE *fp)
{
    float pointbuffer[POINT_BUFFER_SIZE];
    unsigned int num_read, num_requested;

    while (numPoints != 0) {
        if (numPoints > max_vertices) {
            num_requested = max_vertices * 2;
            num_read = readAndSwap((char *)&pointbuffer,
                4, num_requested,fp,DOSWAP);
            if (num_read != num_requested) {
                printf (“error reading SLC data\n”);
                exit(2);
            }
            print_points(pointbuffer, max_vertices);
```

```
                numPoints = numPoints - max_vertices;
            } else if (numPoints != 0) {
                num_requested = numPoints * 2;
                num_read = readAndSwap((char *)&pointbuffer,
                    4, num_requested,fp,DOSWAP);
                if (num_read != num_requested) {
                    printf ("error reading SLC data\n");
                    exit(2);
                }
                print_points(pointbuffer, numPoints);
                numPoints = 0;
            }
        }
}

void process_contour_layers(FILE *fp)
{
    float minLayer;
    long unsigned int numContours, numPoints, numGaps;
    int num_read, i;

    numContours = 0;
    while (numContours != 0xFFFFFFFF) {
        num_read = readAndSwap((char *)&(minLayer),
                sizeof(minLayer),1,fp,DOSWAP);
                num_read = readAndSwap((char *)&(numContours),
                sizeof(numContours),1,fp,DOSWAP);
        if (numContours != 0xFFFFFFFF) {
            printf ("Minumum layer number: %7.3f\n",minLayer);
            printf ("Number of contours: %u\n\n", numContours);
            for (i = 0; i < numContours; i++) {
                printf ("Contour: %d\n", i);
                num_read = readAndSwap((char *)&(numPoints),
                        sizeof(numPoints),1,fp,DOSWAP);
                num_read = readAndSwap((char *)&(numGaps),
                              sizeof(numGaps),1,fp,DOSWAP);
                        printf (“Number of Vertices: %u\n”,numPoints);
                        printf (“Number of Gaps: %u\n”, numGaps);
                        process_points(numPoints,fp);
                        printf(“\n”);
                   }
              } else
                   printf(“Maximum layer number: %7.3f\n”, minLayer);
         }
         printf (“\nEnd of SLC File\n”);
    }

    FILE *openslcFile(char *filename)
    {
         FILE *fp;
         if ((fp = fopen(filename,”rb”)) == NULL) {
              printf (“Unable to open file: %s\n”, filename);
              exit(1);
         }
         printf (“SLC filename: %s\n”, filename);
         return(fp);
    }

    void process_slc_file(char *filename)
    {
         FILE *pslcfile;

         pslcfile = openslcFile(filename);
         read_slc_header(pslcfile);
         read_reserved_section(pslcfile);
         read_table_entry(pslcfile);
         process_contour_layers(pslcfile);
         fclose(pslcfile);
    }
```

```
void main(int argc, char **argv)
{
     char filename[256];
     if (argc > 1) strcpy(filename, argv[1]);
     else {
          printf (“Enter contour file name: “);
          scanf(“%s”,filename);
     }
     process_slc_file(filename);
}
```

CHAPTER 14

AVW & Tcl/Tk

14.1 What is Tcl?

TCL stands for Tool Control Language, a scripting language developed by Dr. John Ousterhout, now at Sun Microsystems. TK extends TCL to support windows on a screen. The BIR has extended TCL further by adding AVW.

TCL has all the features of a programming language, but has a limited set of variable types and is interpreted rather than compiled. TCL is designed to be embedded in another program, but when its embedded in a shell (called tclsh), it becomes an interactive language. This makes it easy to create small, simple programs and get them running. TCL is available for Unix, Wintel, and Mac.

TABLE 14-1 Some TCL language statements, and their C counterparts:

Function	**C**	**TCL**
Declaration	char *a, b[32]; int i,j;	none: all vars are strings
Assignment	strcpy(b,"a string");	set b “a string”
Assignment	a = b;	set a $b
Math Functions	i = sqrt(j + 220);	set i [expr sqrt($j + 220)]
Control Statement	if (i < 200) j = 20;	if { $i < 200 } { set j 20 }

TCL statements can be collected into a file, and this is a TCL script. Functions within a script are created with the “proc” statement, and the function names are then executable. Arguments are passed by position. Functions return values when enclosed in [square brackets]. Functions have a local scope, and all global values are declared with the global proc.

FIGURE 14-1 Tcl Examples

```
#!/bin/tclsh
puts “Hello, World”
```

```
Or;

#!/bin/tclsh
set a “Hello World”
proc show_greeting { message } {
    puts $message
    return “Greeting Sent”
}
set b [show_greeting $a ]
puts $b
```

14.2 What is Tk?

TK adds the ability to create windows and window elements (buttons, menus, etc.) in TCL. The amount of detail involved in writing a program to display a menu in a window, in C, for X Windows or Microsoft Windows can be overwhelming, and most of the time only the default values are used. TK assumes the default values and provides options for other details.

tclsh, when extended with TK, is called *wish*. We have created *AVW_wish4.2*, which is *wish* extended to support all of the AVW functions and structures.

Interactively, when *AVW_wish4.2* is run, a window appears on the screen, and a prompt appears in the shell from which it was started. This allows commands to be typed in the shell window and have the results show up in the wish window.

For an example, lets try:

```
> AVW_wish4.2
% label .lab -text “Hello World”
```

This creates a label widget with the words in it, but it doesn’t put it in the window, because it doesn’t know where to put it. There’s 3 methods for placing objects in wish windows: pack, place and grid. Pack is the simplest.

```
% pack .lab
```

This fits the entire window to the label. Adding and packing another label:

```
% label .lab2 -text “MY, My!”
% pack .lab2
```

14.3 How do AVW and Tcl go together?

The **AVW** functions can be called from a TCL script. For example:

C code:

```
AVW_Image *im;
im = AVW_ReadImageFile(file, NULL);
```

TCL code:

```
set im [AVW_ReadImageFile $file NULL]
```

14.4 What about AVW and TK?

AVW is “displayless”; it has no functions to display or print an image. TK displays images through a widget called a ‘photo’. But the photo widget only supports GIF and JPEG images. So, a bridge, called avw2photo has been implemented. This TCL function copies an AVW image into a photo widget:

```
set Photo [image create photo -palette $ncolors]
avw2photo $im $Photo
```

14.5 Imaging Program Pieces: Loading an Image

Let’s assume that we have an image file stored on disk somewhere. We’re also going to assume that it was stored in one of the 30-odd formats that **AVW** reads. Before the file can be read, it has to be opened. We use an **AVW_OpenImageFile**:

```
set filename “avw_logo.gif”
set imf [AVW_OpenImageFile $filename “r”]
```

The “r” indicates that we’re going to open this file only for reading. The *imf* returned is a handle to the image file that is used in later operations to read the image. If the open fails for some reason (there’s no such file, or it isn’t an image file) then an error will be generated and the program will end.

The information in the **AVW** structures, such as **AVW_ImageFile**, is also available. The **AVW** C structure can be interrogated by using the imf as a function name:

```
set w [$imf Width]
set h [$imf Height]
set d [$imf Depth]
```

How the image data is read will depend on the nature of the file. If it’s an image, the following will work:

```
set im [AVW_ReadImageFile $imf NULL]
```

If the data is 3d, the entire volume can be read:

```
set imv [AVW_ReadVolume $imf NULL]
```

Note that doing this only makes the pixels or voxels in the file accessible to the functions in the program.

14.6 Displaying an Image

Now that we’ve opened our file and read an image from it, we want to display it on the screen. First, we’ll need to create a TCL image object to display the image on the screen:

```
set p [image create photo -palette 256/256/256]
```

This creates a 24 bit photo type image. An 8 bit photo, for for greyscale images, can be created by using a palette of 256, instead of 256/256/256. We need to set the size of our photo to the same size as our image. Note this could have been done in the same command which created the photo:

```
$p configure -width [$im Width] -height [$im Height]
```

Then we place our **AVW_image** into our photo image:

```
avw2photo $im $p
```

So far, all we’ve created is a TCL photo object. We haven’t placed it on the screen. To do that, we’ll create a canvas, which is a TK object for holding all kinds of objects, including image objects.

```
canvas .can -width [$im Width] -height [$im Height]
```

```
pack .can
```

Then we'll create an image space on the canvas:

```
set cimage [.can create image 0 0]
```

Now we'll map the photo onto the canvas image:

```
.can itemconfigure $cimage -image $p
```

Our image should now be displayed in the window. The only problem is that the image is not fully displayed... it's shifted up and left. We solve this by anchoring the image in the canvas:

```
.can itemconfigure $cimage -anchor nw
```

We can condense much of this by applying all the options at creation time:

```
set p [image create photo -palette 256/256/256 -width [$im Width] -height
[$im Height]]
pack [canvas .can -width [$im Width] -height [$im Height]]
set cimage [.can create image 0 0 -image $p -anchor nw]
```

14.6.124 bit displays.

To get full color images on 24 bit displays, you need to set the visual to truecolor. Most systems provide a default pseudocolor visual. Visuals can only be set in a frame, a tk object which serves as a placeholder for other objects. To do this, first create the frame with the proper visual, then make the canvas a member of the frame:

```
frame .f -visual truecolor
canvas .f.can -width [$im Width] -height [$im Height]
```

14.7 Adding AVW Image Processing Functions

OK, we've opened a file and loaded an image and placed it in a window. What about using all those **AVW** functions on the image? The *im* is the reference to our original image. Most **AVW** functions include a "reuse" argument which allows the function to replace the image with the results of the function. We can use this, but we'll avoid using it the first time. That way, we'll retain the original image if we wish to do other functions on it.

Let's start with something simple, such as resizing the image and redisplaying it:

```
set im2 [AVW_ResizeImage $im 89 209 AVW_TRUE NULL]
```

Now, we want to re-display the results of our processing. First, we’ll resize our photo and canvas to the new size of the image. We’ll use variables here to speed things up:

```
set w [$im2 Width]
set h [$im2 Height]
.can configure -width $w -height $h
$p configure -width $w -height $h
```

Next, we change the image:

```
avw2photo $im2 $p
```

Since the starting image used a lookup table, and the interpolation created pixels that were averages of pixel values, the resulting display appears erroneous. We can correct this by using nearest neighbor lookup:

```
set im2 [AVW_ResizeImage $im 89 209 AVW_FALSE NULL]
```

then remapping the image to the display:

```
avw2photo $im2 $p
```

14.8 Interacting with Boxes and Buttons

To interact with the window we need to add some TK widgets.

First, lets add a label to tell us the filename:

```
pack [label .lab -text $filename]
```

Then, let’s add a button to display our original image and a button to display our resized image. Since the code to change the images is multi-line, we’ll make each into a proc

```
proc display_orig { } {
global im p
set w [$im Width]
set h [$im Height]
.can configure -width $w -height $h
$p configure -width $w -height $h
avw2photo $im $p
}
```

```
proc display_resized { } {
global im2 p
set w [$im2 Width]
set h [$im2 Height]
.can configure -width $w -height $h
$p configure -width $w -height $h
avw2photo $im2 $p
}
```

Now, we can create our buttons, and have them call the correct procedures:

```
pack [button .b1 -text “Original” -command display_orig]
pack [button .b2 -text “Resized” -command display_resized]
```

The two procs we have created are very similar. In fact, most programmers would write this a bit more compactly:

```
proc display_image { i } {
global p
set w [$i Width]
set h [$i Height]
.can configure -width $w -height $h
$p configure -width $w -height $h
avw2photo $i $p
}
```

Then buttons can now be defined with:

```
pack [button .b1 -text “Original” -command “display_image $im”]
pack [button .b2 -text “Resized” -command “display_image $im2”]
```

14.9 Adding a C function to your Program

Tcl can be extended to support new image processing algorithms provided that the C code and a library containing the algorithm are available.

The function must be set up as a callable C function (it can't be an entire program). Some C wrapping to handle the TCL API needs to be added.

Let's assume that the function takes 3 arguments, the width and height of an image, and a memory pointer to the actual image pixels. And, we'll restrict the function to operating on 8 bit unsigned char pixel values. It returns an integer value. In C, the function is declared:

```
int myfunc(int w, int h, unsigned char *mem){
        (body of function);
}
```

In TCL, this function will be called with the following statement:

```
set r [myfunc $w $h $im]
```

where w and h are width and height, and im is an AVW image. The following example supplies the necessary code to make this work:

```
#include “AVW.h”
#include “tcl.h”
#include “AVWTCL.h”

int myfunc_command(ClientData clientData, Tcl_Interp *interp, int argc,
char **argv)
{
    avwtcl_Image *ImagePtr;
    int w,h,i;
    if(Tcl_GetInt(interp, argv[1], &w)) return(TCL_ERROR);
    if(Tcl_GetInt(interp, argv[2], &h)) return(TCL_ERROR);
    if(!(ImagePtr = avwtcl_GetImage(interp, argv[3], AVW_FALSE)))
    {
        return(TCL_ERROR);
    }
    i = myfunc(w,h,ImagePtr->Image->Mem);
    sprintf(interp->result, “%d”, i);
    return(TCL_OK);
}

int Myfunc_Init(Tcl_Interp *interp)
{
    Tcl_CreateCommand(interp,”myfunc”, myfunc_command, (ClientData)
NULL,         (Tcl_CmdDeleteProc *)NULL);
}
```

Save this code into a file called “Myfunc_wrap.c”. and assume the function source code is in the file “myfunc.c”. These two files need to be compiled into object files, and then shared library must be built from them. For example, on a Sun system:

```
cc -c Myfunc_wrap.c myfunc.c -I$AVW/include -L$AVW/$TARGET/lib -lAVW -lm
ld -G -o Myfunc.so Myfunc_wrap.o myfunc.o
```

On an SGI system:

```
cc -c MyFunc_wrap.c myfunc.c -I$AVW/include -L$AVW/$TARGET/lib -lAVW -lm
ld -shared -o Myfunc.so Myfunc_wrap.o myfunc.o
```

After starting **AVW_wish4.2** the library needs to be loaded :

```
% load Myfunc.so
```

Note that the name of the library and the name of the Init function must match.

14.10 Saving an Image

Once all of the image processing is done it is usually desirable to save the results. For simplicity, we’ll assume that the image to be saved is a 2D image, in a TCL variable called im3.

First, the image file in which we'll save the results needs to be created. A file format needs to be selected. The formats which **AVW** supports for writing can be obtained as follows:

```
set flist [avw_l2str [AVW_ListFormats AVW_SUPPORT_WRITE]]
```

This list can be printed out interactively or a TK menu or listbox can be created to allow selection from the window. For our program, we'll select one, JPEG. Now the image file can be created:

```
set imout [AVW_CreateImageFile “outfile.jpg” JPG [$im3 Width] [$im3
Height] 1 [$im3 DataType]]
```

Then image can now be written to the file:

```
AVW_WriteImageFile $imout $im3
```

For completeness the files should be closed:

```
AVW_CloseImageFile $imf
AVW_CloseImageFile $imout
```

14.11Sample Code

OK, now we've explored the entire process from beginning to end. What we've left out is many details about TK options, like where to place objects in a canvas, and how to set background colors, etc. We've also left out all the error checking. Remember, we just wanted to get you started with AVW/TCL in this chapter.

The example tcl script (myfunc_ex.tcl) and C code (myfunc.c Myfunc_wrap.c) and another example tcl script (example.tcl) which opens an image file, displays a slice, then puts up a slider to step through the slices of a volume image are located in **$AVW/extras/tcl** directory. There are also example tcl scripts in many of the subdirectories of **$AVW/extras.** These scripts correspond to the example C programs.

CHAPTER 15 *AVW++*

The Biomedical Imaging Resource has developed a C++ version of **AVW** (currently referred to as **AVW++**) in the form of a class library. In general, classes are based on the **AVW** structures and the methods are based on the **AVW** functions. These functions have not been rewritten but rather are called directly from within the methods, making this library a C++ “wrapper” for **AVW**.

The member data for most of the classes consists only of the corresponding **AVW** structures. This construct provides a layer of protection against direct manipulation of the data, i.e. data encapsulation. Each of the **AVW** functions has been matched with the appropriate class and is called via public methods of the same name or through the constructors, destructors and operator overload methods (=, +, -, *, /, ==, etc.).

AVW++ was developed to support internal C++ projects, and does not have the same documentation and support found in **AVW**. It is, however, recommended that any C++ developers requiring **AVW** functionality consider looking into **AVW++** first.

15.1 Classes

The following classes are found in **AVW++**:

FIGURE 15-1 **AVW++ Classes**

`A_Colormap`	`A_ObjectMap`
`A_FPoint2`	`A_Point2`
`A_FPoint3`	`A_Point3`
`A_FPointList2`	`A_PointList2`
`A_FPointList3`	`A_PointList3`
`A_FilterCoeffs`	`A_PointValueList`
`A_Histogram`	`A_Rect2`
`A_Image`	`A_Rect3`

`A_ImageFile`	`A_RenderedImage`
`A_IntensityStats`	`A_Surface`
`A_Line2`	`A_Volume`
`A_Line3`	`A_VolumeRenderParms`
`A_Matrix`	`K_String`

15.2 Example Code

The following is sample C++ code fragments using **`AVW++`**:

FIGURE 15-2 AVW++ Sample Code

```
.
.
.
{
        K_String fileName("/images/demos/sample.hdr");          // creates file name string
        A_ImageFile myImageFile(fileName);                      // opens specified image file
        A_Image currImage(myImageFile.getImage(4, 2));          // constructor: slice 4, vol 2
        A_Image oldImage = currImage;                           // makes a copy of currImage

// the following flips currImage vertically, resizes it 4x, then dithers it to 64 colors
        currImage.flip(AVW_FALSE, AVW_TRUE).resize(400,400).dither(64);

        .
        .       // 2 other A_Image's called otherImage and newImage are constructed.
        .

// the following adds two images if they are not the same
        if (currImage != otherImage)                            // != operator is overloaded
        newImage = currImage + otherImage;                      // + operator is overloaded
        .
        .       // an AVW_Volume* called anAVWVolume is created
        .
        A_Volume someVolume(anAVWVolume);                    // constructor: directly from an AVW_Volume*

// the following puts currImage into someVolume oriented coronally at slice 4
        someVolume.putOrthogonal(currImage, AVW_CORONAL, 4);
        .
        .
        .
}               // Note: at this point the destructor for each object constructed
                // within the scope of these brackets is called, freeing allocated
                // memory automatically, thus simplifying memory management.
.
.
.
```

To inquire about acquiring **AVW++** contact the Biomedical Imaging Resource, Mayo Foundation.

List of Figures

FIGURE 2-1 AVW_Image Structure 5
FIGURE 2-2 Memory Type Casting 6
FIGURE 2-3 Data Types From AVW.h 6
FIGURE 2-4 AVW_Colormap Structure From AVW.h 7
FIGURE 2-5 Info String Example 8
FIGURE 2-6 AVW_Volume Structure 9
FIGURE 2-7 Create and Destroy Example 10
FIGURE 2-8 Image Recycling 11
FIGURE 2-9 Image Reuse Example 12
FIGURE 2-10 Point Structures 13
FIGURE 2-11 Point List Structure 14
FIGURE 2-12 Point List Example 14
FIGURE 3-1 The AVW_ImageFile Structure 18
FIGURE 3-2 The AVW_ExtendIO structure 19
FIGURE 3-3 AVW I/O Functions 20
FIGURE 3-4 Image File Format Properties 21
FIGURE 3-5 Listing File Format Names by Supported Properties 22
FIGURE 3-6 Determining if a File Format Accepts a Floating Point Image 22
FIGURE 3-7 Setting the Voxel Dimensions of an Image File 23
FIGURE 3-8 Info String Entries From an Interfile Format Image File 24
FIGURE 3-9 Supported AVW Image File Formats 24
FIGURE 3-10 Example of an AVW_VolumeFile 25
FIGURE 3-11 Opening an AVW_VolumeFile with Wild Card File Specification 26
FIGURE 3-12 Writing Images to an AVW_VolumeFile and Individual SunRaster Files 26
FIGURE 3-13 Creating a SubVolume Description File 27

FIGURE 3-14 Text Contents of an AVW_ImageFile 28
FIGURE 3-15 Reading Data into an AVW_Image 29
FIGURE 3-16 Opening and reading images from an UNKNOWN file: 30
FIGURE 3-17 I/O Extension Functions 31
FIGURE 3-18 Extending AVW to Support TIFFAVW_ImageFile *avw_open_tiff_image(); 32
FIGURE 3-19 Example I/O Programs 33
FIGURE 3-20 tcl Image I/O example code 34
FIGURE 4-1 Excerpts from AVW.h 37
FIGURE 4-2 Orthogonal Plane Example 38
FIGURE 4-3 Oblique Plane Example 39
FIGURE 4-4 Curved Plane Example 40
FIGURE 4-5 Erasing RGB Image Example 41
FIGURE 4-6 DataTypes from AVW.h 42
FIGURE 4-7 Convert to RGB 42
FIGURE 4-8 Thresholding Example 43
FIGURE 4-9 Intensity Scaling Example 43
FIGURE 4-10 Extract the Center of an Image 46
FIGURE 4-11 Magnify an Image 4 Times Using Replication 47
FIGURE 4-12 AVW Mathematical Operators 47
FIGURE 4-13 Add and Average All Images in a 16-bit Volume 50
FIGURE 4-14 AVW_Matrix 50
FIGURE 4-15 Rotate an Image 45 Degrees Counter Clockwise Around Its Center. 52
FIGURE 5-1 AVW_Histogram Structure from AVW_Histogram.h 54
FIGURE 5-2 Match Image to Given Histogram 54
FIGURE 5-3 Match Volume Histogram to Image 55
FIGURE 5-4 Flatten Image Histogram 55
FIGURE 5-5 Preserve Volume Histogram 56
FIGURE 5-6 Convolve Image Example 1 57
FIGURE 5-7 Convolve Image Example 2 58
FIGURE 5-8 Convolve Volume Example 58
FIGURE 5-9 5 X 3 Lowpass Filter 59
FIGURE 5-10 5X3 Enhanced Unsharp Filter 60
FIGURE 5-11 3 X 3 Sigma Filter 62

FIGURE 5-12 AVW_SigmaFilterImage Results 63
FIGURE 5-13 VSF Mean Filter 64
FIGURE 5-14 VSF Mean Filter Results 64
FIGURE 5-15 Rank Filter 65
FIGURE 5-16 Rank Filter Results 66
FIGURE 6-1 FFT and Convolution Example 67
FIGURE 6-2 Gaussian Filter Example 68
FIGURE 6-3 Wiener Deconvolution 70
FIGURE 7-1 Threshold Example 73
FIGURE 7-2 Four and Eight Connected Neighbors 74
FIGURE 7-3 Six and Twenty Six Connected Neighbors 74
FIGURE 7-4 Neighbor Definitions from AVW.h 75
FIGURE 7-5 Segmentation Example 77
FIGURE 8-1 Line Profile Example 79
FIGURE 8-2 Make Mask Example 80
FIGURE 8-3 Histogram Collection 81
FIGURE 8-4 Intensity Statistics Example 82
FIGURE 8-5 AVW_IntensityStats Structure 85
FIGURE 8-6 AVW_2DShapeStats Structure 86
FIGURE 8-7 Shape Measurement Example 87
FIGURE 8-8 3D Measurement Example 87
FIGURE 9-1 AVW_RenderParameters Structure 91
FIGURE 9-2 Initialization Example 92
FIGURE 9-3 Render Types Defined in AVW_Render.h 93
FIGURE 9-4 Delete Depth Definitions in AVW_Render.h 95
FIGURE 9-5 360 Degree Rotation Sequence 96
FIGURE 9-6 Screen Coordinates to Render Space 99
FIGURE 10-1 AVW_ObjectMap structure defined in AVW_ObjectMap.h 103
FIGURE 10-2 AVW_Object Structure 104
FIGURE 10-3 Creating and Deleting an Object Map 107
FIGURE 10-4 Saving an Object Map 107
FIGURE 10-5 Object Map Version Numbers 110
FIGURE 10-6 Loading an Object Map 111
FIGURE 10-7 Using a Volume to Define a New Object 112
FIGURE 10-8 Using a region growing to define a new Object 113

FIGURE 10-9 Using AVW_Volume/AVW_Image Functions with Object Maps 114
FIGURE 11-1 Histogram from a Classified Image 116
FIGURE 11-2 Three Bands from an RGB Image with Training Image 117
FIGURE 11-3 Classifying an Image 117
FIGURE 11-4 Classifying a volume 118
FIGURE 11-5 Preparing a Training Image 119
FIGURE 11-6 Classify Volume Control Parameters 120
FIGURE 11-7 Scattergram Example 121
FIGURE 11-8 Making a Scattergram 122
FIGURE 11-9 A Processed Scattergram 122
FIGURE 11-10 Classifying Images with a Processed Scattergram 122
FIGURE 11-11 Classified Image 123
FIGURE 12-1 AVW_MatchParameters Structure From AVW_SurfaceMatch.h 125
FIGURE 12-2 AVW_MatchResult Structure from AVW_SurfaceMatch.h 127
FIGURE 12-3 AVW_MatchSurfaces() Example 127
FIGURE 13-1 AVW_TileParameters Structure 136
FIGURE 13-2 Initialization Example 137
FIGURE 13-3 Tile Types Defined in AVW_Model.h 138
FIGURE 13-4 Calculation of Local Curvature 139
FIGURE 13-5 Major Axis Types Defined in AVW.h 139
FIGURE 13-6 Shape Orientation Types Defined in AVW.h 140
FIGURE 13-7 Kohonen Network Flags Defined in AVW_Model.h 140
FIGURE 13-8 Neighborhoods Defined in AVW_Model.h 141
FIGURE 13-9 AVW_TiledSurface Structure Defined in AVW_Model.h 143
FIGURE 13-10 Creating and Deleting a Tiled Surface 144
FIGURE 13-11 Saving a Tiled Surface 145
FIGURE 13-12 Supported Output File Formats Defined in AVW_Model.h 145
FIGURE 13-13 Loading a Tiled Surface 147
FIGURE 13-14 Supported Input File Formats 147
FIGURE 13-15 Supported Contour File Formats Defined in AVW_Model.h 148
FIGURE 13-16 Extracting Contours 149

FIGURE 13-17 The AVW_RPParam Structure 150
FIGURE 13-18 dumpslc.c: sample code to dump slc formatted files to ASCII 151
FIGURE 14-1 Tcl Examples 159
FIGURE 15-1 AVW++ Classes 169
FIGURE 15-2 AVW++ Sample Code 170

Index

Numerics

24-bit RGB 12

A

ACRNEMA 24
AddNodeFreq 142
Analysis 2, 79
ANALYZE 104, 105
AnalyzeImage 23, 24, 28
AnalyzeScreen 24
ANSI C Prototyping 16
Area 86
area 2
AutoCAD 146
autotype 120
AVW 17, 20, 58, 79
AVW/TCL 168
AVW_26_CONNECTED 75
AVW_2DShapeStats 86, 87
AVW_2DShapeStats Structure 86
AVW_4_CONNECTED 75
AVW_6_CONNECTED 75
AVW_8_CONNECTED 75
AVW_AddObject 112, 113, 114
AVW_AddPoint2 15, 80
AVW_AddPoint2(40
AVW_AddPoint2() 40
AVW_BUBBLE_NEIGHBORHOOD 142
AVW_ClassifyFromScattergram 122
AVW_ClassifyImage 117, 118, 119
AVW_ClassifyVolume 117, 118, 120
AVW_ClearHistogram 82
AVW_CLIP_ACTUAL 97
AVW_CLIP_REMOVE_AND_RENDER 97
AVW_CLIP_RENDER_AS_IS 97
AVW_CLIP_SHADED 97
AVW_CloseImageFile 17, 20, 23, 58, 129
AVW_COLOR 6, 12, 44
AVW_ColorMap 7
AVW_Colormap 7, 12, 44
AVW_COMPLEX 6, 67
AVW_Compute2DShapeStats 87
AVW_ComputeCircularity 87
AVW_ComputeImageCentroid 87
AVW_ComputeImageIntensityStats 83
AVW_ComputeLineProfile 79, 80
AVW_ComputeMEB 88, 89
AVW_ComputeMER 87
AVW_ComputePerimeter 87
AVW_ComputeRFF 87
AVW_ComputeVolume 88, 89
AVW_ComputeVolumeCentroid 88, 89
AVW_ComputeVolumeIntensityStats 84
AVW_ConditionalDilateImage 76
AVW_ConditionalDilateVolume 76, 77
AVW_ConnectAndKeepVolume 75
AVW_ConstantOpImage(49
AVW_ConvertImage 50
AVW_ConvolveImage 58
AVW_ConvolveVolume 59
AVW_CopyImage() 40
AVW_CopyMatrix 51
AVW_CORONAL 37, 140
AVW_CreateButterworthCoeffs 68
AVW_CreateCircularMTF 68
AVW_CreateCoeffs 68
AVW_CreateGaussianCoeffs 68
AVW_CreateHistogram 54, 82
AVW_CreateImage 48, 76, 119
AVW_CreateImage() 9, 12, 49
AVW_CreateImageFile 17, 18, 20, 26
AVW_CreateObjectMap 106, 107
AVW_CreatePoint2List 79
AVW_CreatePointList2 14, 80

AVW_CreatePointList2() 40
AVW_CreatePointList3 113
AVW_CreateSphericalMTF 68
AVW_CreateStoksethMTF 68
AVW_CreateStructuringImage 76
AVW_CreateStructuringVolume 76, 77
AVW_CreateVolume 58, 76
AVW_CreateVolume() 9
AVW_DeconvDivideImage 69
AVW_DeconvDivideVolume 69
AVW_DeconvWienerImage 70
AVW_DeconvWienerVolume 70
AVW_DEFINE_ALL_THE_WAY 95
AVW_DEFINE_SINGLE_LAYER 95
AVW_DEFINE_SINGLE_VOXEL 95
AVW_DefineConnected 113, 114
AVW_DELETE_ALL_THE_WAY 95
AVW_DELETE_SINGLE_LAYER 95
AVW_DELETE_SINGLE_VOXEL 95
AVW_DELETE_VOXELS 93, 95, 113
AVW_DeleteObject 112
AVW_DEPTH_SHADING 93, 94
AVW_DestorySurface 100
AVW_DestroyHistogram 55
AVW_DestroyImage 55
AVW_DESTROYIMAGE() 10
AVW_DestroyImage() 10
AVW_DestroyObjectMap 106, 111, 113
AVW_DestroyPointList2 15
AVW_DestroyPointList3 114
AVW_DestroyRenderedImage 101
AVW_DestroyRenderParameters 101
AVW_DestroyTiledSurface 144
AVW_DESTROYVOLUME() 10
AVW_DestroyVolume() 10
AVW_DilateImage 76
AVW_DilateVolume 76
AVW_DitherImage 44
AVW_DrawRenderedLine 100
AVW_DrawRenderedPoint 100
AVW_DrawTiledSurface 147
AVW_DXF_SURFACE 145, 146, 147
AVW_ErodeImage 76
AVW_ErodeVolume 76
AVW_Error 82, 84, 87
AVW_ErrorMessage 84, 87
AVW_ErrorNumber 84, 87
AVW_Errors.h 16
AVW_ExtendI/O 21
AVW_ExtendImageFile 20
AVW_ExtendIO 19, 31
AVW_ExtractSurface 100
AVW_F_ABS 48, 49
AVW_F_AVG 48, 49
AVW_F_COUNT 47, 49
AVW_F_LOG 47, 48
AVW_F_MAX 48, 49
AVW_F_MIN 48, 49
AVW_F_SQRT 47, 48
AVW_F_XPOS 47, 49
AVW_F_YPOS 48, 49
AVW_FAIL 84
AVW_FFT2D 68, 69, 70
AVW_FFT3D 67
AVW_FillHolesImage 77
AVW_FilllHolesVolume 77
AVW_FilterCoeffs 68
AVW_FindImageComponents 75
AVW_FindImageMaxMin 44
AVW_FindRenderedPoint 99, 100
AVW_FindVolumeComponents 75, 77
AVW_FindVolumeEdges 77
AVW_FlattenHistogramImage 55
AVW_FlattenImageHistogram 56
AVW_FlipImage() 41
AVW_FLIPX 41
AVW_FLIPY 41
AVW_FLOAT 6, 67
AVW_FormatSupports 20, 22
AVW_FPoint2 13
AVW_FPoint3 14, 88
AVW_FunctionImage 48
AVW_GAUSS_NEIGHBORHOOD 142
AVW_GAUSSIAN_CLUSTER 117, 118, 120
AVW_GetCurved() 40
AVW_GetImageHistogram 55, 82, 116
AVW_GetNumericInfo 23
AVW_GetNumericInfo() 7
AVW_GetObject 112
AVW_GetOblique 39
AVW_GetOblique() 39
AVW_GetOrthognal() 38
AVW_GetOrthogonal 27, 55, 58, 114
AVW_GetOrthogonal() 12
AVW_GetStringInfo() 7
AVW_GetSubImage 46
AVW_GetThresholdedBoundary 75
AVW_GetVolumeHistogram 81, 82

AVW_GRADIENT_SHADING 93, 94
AVW_Histogram 54, 81
AVW_Image 5, 7, 9, 17, 23, 38, 85
AVW_ImageFile 17, 18, 23, 24, 25, 28, 162
AVW_ImageFile Format 28
AVW_ImageOpConstant 49, 50
AVW_ImageOpConstant() 49
AVW_ImageOpImage 50, 67, 68, 69
AVW_InitializeRenderParameters 92, 95, 96, 98
AVW_InitializeTileParameters 137, 139
AVW_InitializeTileParamters 144
AVW_IntensityScale 44
AVW_IntensityScaleImage 44
AVW_IntensityStats 83, 84, 85
AVW_INVENTOR_SURFACE 145, 146
AVW_InvertImage 45
AVW_InvertMatrix 51, 127
AVW_IPoint2 13
AVW_IPoint3 14
AVW_IterDeconvImage 71
AVW_IterDeconvVolume 71
AVW_K_NEAREST_NEIGHBOR 120
AVW_LabelImageFromEdges 77
AVW_LabelVolumeFromEdges 77
AVW_ListFormat 168
AVW_ListFormats 19, 20, 22
AVW_LoadObjectMap 104, 111, 112
AVW_LoadTiledSurface 146
AVW_LowpassFilterVolume 59, 60
AVW_MakeGrayImage 45
AVW_MakeMaskFromTrace 80
AVW_MakeVolumeFromImage() 45
AVW_MARCHING_CUBES 138
AVW_MatchImageHistogram 54, 55
AVW_MatchParameters 125, 126, 127
AVW_MatchResult 125, 126
AVW_MatchResults 125
AVW_MatchSurface 127
AVW_MatchSurfaces 125, 127, 129
AVW_MatchSurfaces() 125
AVW_Matrix 98, 127
AVW_MAX_INTENSITY_PROJECTION 93, 94
AVW_Measure. 86
AVW_Measure.h 79
AVW_MirrorMatrix 51
AVW_MirrorRendered 99
AVW_MorphCloseImage 76
AVW_MorphCloseVolume 76
AVW_MorphMaxImage 76
AVW_MorphMinImage 76
AVW_MorphOpenImage 76
AVW_MorphOpenVolume 77
AVW_MultiplyMatrix 51
AVW_NEAREST_NEIGHBOR 120
AVW_NearestNeighborDeconv 71
AVW_NEURAL_NETWORK 118, 120, 121
AVW_NonMaxImage 76
AVW_NonMaxVolume 76
AVW_OBJ_SURFACE 145, 146, 147
AVW_Object 104
AVW_Object Structure 104
AVW_ObjectMap 103, 104
AVW_OP_ADD 47
AVW_OP_AND 47
AVW_OP_DIV 47
AVW_OP_EQ 48
AVW_OP_GE 47
AVW_OP_GT 47
AVW_OP_LE 47
AVW_OP_LT 47
AVW_OP_MOD 48
AVW_OP_MUL 47
AVW_OP_NE 48
AVW_OP_OR 47
AVW_OP_SUB 47
AVW_OpenImageFile 17, 18, 20, 26, 58, 128, 129, 161
AVW_OrthoGradFilterVolume 62
AVW_PadImage 46
AVW_PadImage() 46
AVW_PARZEN_WINDOW 120
AVW_PARZEN_WINDOWS 120
AVW_PERSPECTIVE_FLOA 98
AVW_PERSPECTIVE_FLOAT 98
AVW_PERSPECTIVE_INT 98
AVW_PERSPECTIVE_OFF 98
AVW_Point2 13, 79
AVW_Point3 13
AVW_PointList2 14, 79
AVW_PointValueList 79
AVW_POLY_SURFACE 145, 146, 147
AVW_PreserveVolumeHistogram 56
AVW_ProcessZGradients 94, 99
AVW_PutNumericInfo 23, 85
AVW_PutNumericInfo() 7
AVW_PutObject 112, 113
AVW_PutOblique 39
AVW_PutOblique() 39
AVW_PutOrthogonal 55, 58, 114

AVW_PutOrthogonal() 38
AVW_PutPixel 119
AVW_PutStringInfo 85
AVW_PutSubImag 46
AVW_PutVoxel 76
AVW_RankFilterImage 65
AVW_ReadImageFile 17, 20, 162
AVW_ReadSwap 29, 30
AVW_ReadVolume 20, 58, 128, 162
AVW_RemovePoint2 15
AVW_RenderedImage 92, 96
AVW_RenderParameters 91, 92, 96
AVW_RenderSurface 100
AVW_RenderVolume 91, 92, 96, 99, 104, 113
AVW_ResizeImage 47, 163, 164
AVW_ResizeVolumeSliceBySlice 47
AVW_ReverseBits 29
AVW_RotateMatrix 39, 51, 96
AVW_SAGITTAL 37, 140
AVW_SaveObjectMap 104, 107, 113
AVW_SaveTiledSurface 145, 146
AVW_ScaleMatrix 51
AVW_Scattergram 122
AVW_SeekImageFile 17, 20, 27
AVW_SetIdentityMatrix 39, 51, 96
AVW_SetImage() 40
AVW_ShiftImage() 41
AVW_SigmaFilterImage 62
AVW_SIGNED_CHAR 6
AVW_SIGNED_INT 6
AVW_SIGNED_SHORT 6
AVW_SobelFIlterEnhanceVolume 61
AVW_SobelFilterVolume 61
AVW_STL_SURFACE 145, 146, 147
AVW_SUCCESS 84
AVW_SUMMED_VOXEL_PROJECTION 93, 94
AVW_SURFACE_PROJECTION 93, 94
AVW_SurfaceMatch.h 125
AVW_SwapBlock 29
AVW_SwapDouble 30
AVW_SwapFloat 30
AVW_SwapInt 30
AVW_SwapLong 30
AVW_SwapShort 30
AVW_TableImage 45
AVW_Thin2D 77
AVW_Thin3D 77
AVW_ThresholdImage 43, 73, 82, 83, 87
AVW_ThresholdVolume 73, 77, 88
AVW_TILE_GROW 138, 139, 141
AVW_TILE_KOHONEN 138, 139, 141
AVW_TiledSurface 143, 146
AVW_TileParameters 136
AVW_TileSurface 142
AVW_TileVolume 136, 144
AVW_TRAIN_IN_MAJOR_AXIS 140, 141
AVW_TRAIN_WITH_WEIGHT 140, 141
AVW_TransformImage 52
AVW_TransformVolume 52, 127
AVW_TransformVolumeSliceBySlice 52
AVW_TranslateMatrix 51
AVW_TRANSPARENCY_SHADING 93, 95
AVW_TRANSVERSE 37, 140
AVW_TRIANGLE_NEIGHBORHOOD 142
AVW_UltimateErosionImage 76
AVW_UltimateErosionVolume 76
AVW_UnsharpFilterEnhanceVolume 60, 61
AVW_UnsharpFilterVolume 60
AVW_UNSIGNED_CHAR 6, 75, 116
AVW_UNSIGNED_INT 6
AVW_UNSIGNED_SHORT 6
AVW_Volume 9, 85, 112
AVW_VolumeFile 23, 24, 25, 26, 27
AVW_VolumeFile f 27
AVW_VolumeToSLC 148
AVW_VRIO_SURFACE 145
AVW_VRIO_SURRFACE 146
AVW_VRML_SURFACE 145, 146, 147
AVW_wish4.2 160
AVW_WriteImageFile 17, 20, 23, 27
AVW_WriteSubvolDes 27
AVW_WriteSubVolumeDescription 27
AVW_WriteSwap 30
AVW_WriteVolume 20
AVW_XAXIS 139
AVW_YAXIS 140
AVW_ZAXIS 140
avw2photo 161, 162, 164

B

BAP 86
best matching unit 133
best matching units 134
BlockSize 143
BMP 24
Boolean 48
Butterwort 68
BytesPerImage 7

BytesPerLine 7
BytesPerPixel 7
BytesPerVolume 9

C

CAD 146
CAD formats 145
Circularity 86
Classifying a volume 118
Clip Volume 97
ClipHighX 97
ClipHighY 97
ClipHighZ 97
ClipLowX 97
ClipLowY 97
ClipLowZ 97
ClipPlaneMaximum 97
ClipPlaneMinimum 97
ClipShading 97
close() 31
CloseSrfcFlag 139
Color Images 12
Color Lookup Table 7
Colormap 7
Competitive Hebbian Learning 133, 134, 138
Compiling and Linking 16
Constrained Iterative Deconvolution 70
contours 133
Conversion to Gray Scale 45
convolution 57
Convolve 57
Convolve Volume Example 58
coordinate system 15
Coords 143
Coronal 37
Create and Destroy Example 10
create() 31
CTI 24
Curved plane example 40
CurveOpRadius 139

D

d Sagittal 37
Data Types 6
DataFormat 26
DataType 6, 48, 49
deblurring 69
Deconvolution 69
deconvolution 67
DeleteDepth 95
DeleteValue 95
Depth 9
depth gradient shading 3
depth-only shading 3
DepthShaded 99
DICOM 31
direct topological neighbor 135
direct topological neighbors 135
DisplayFlag 105
Dithering 44

E

Eb 143
En 143
EndBlue 105
EndGreen 105
EndRed 105
Enhanced Unsharp Filter 60
Epochs 118, 121
Error Handling 16
Extents 62
EyePosition 98

F

Fast Fourier Transforms 67
FFT 2, 67, 69
FFT and Convolution Example 67
Flipping 41
Four and Eight Connected Neighbors 74
Fourier 67
fractal signature 2
Frequency Space Convolution 67
Function Specification and Calling Conventions 15

G

Gaussian 68, 71
Gaussian Cluster 115
Gaussian Clustering 116
Gaussian Filter Example 68
GE Advantage 22, 24
GE Advantage CT 17
GE9800 24
geometric surface 133
GESigna 24
GESTARCAM 24
GIF 161
GradientShading 100

grayscale 43
growing cell structures 134
Growing Net 135, 136
growing neural gas 134
GrowingAlpha 143
GrowingD 143

H

Height 7
HiddenEpochs 118, 121
HighestPoint 85
Histogram 53
histogram 116
Histogram Flattening 55
Histogram Matching 53
Histogram Preservation 56
Histograms 81

I

Image I/O 2
Image Memory 6
Image Recycling 11
Imatron 24
Indices 143
inflection 139
Info 7, 23, 25, 28, 85
integrated surface projection 3
Intensity Scaling 43
Intensity Scaling Example 43
INTERFILE 24
Interfile 21
interpolation 39
Inventor 145, 146
Inverting 45
isotropic 46

J

JPEG 161, 168
JPG 31

K

K-Nearest Neighbor 116
K-Nearest Neighbor, 115
Kohonen 135
Kohonen Net 138
Kohonen Network 134
Kohonen Networks 133
KohonenAlpha 142
KohonenFlag 140
KohonenMajorAxis 139
KohonenNeighborhood 141
KohonenNeighborRadius 142
KohonenRepetitions 141
KohonenShapeOffset 140
KohonenShapeOrient 140
KValue 118, 120
Kvalue 118

L

Lighting 98
LightMatrix 98
lists 14
Logical 48
LowestIntensity 86

M

man 16
MANPATH 16
Marching Cubes 133
marching cubes 138
MaskValue 95, 98
Mathematical Morphology 75
Matrix 50, 98
MaxDist 116, 118, 120
maximum intensity projection 3
MaximumAge 142
MaximumNodes 143
MaximumPixelValue 98, 104
mea 2
MEB 87, 89
Memory Type Casting 6
Memory Usage 11
MER 86
MERAngle 86
MERAspect 86
MinimumPixelValue 98, 104
Modeling 133
morphological processing 2
multi-spectral analysis 115
Multi-Spectral Classification 115
Multispectral classification 73
multispectral classification 2
multispectral classification. 2

N

Nearest Neighbor 115

nearest neighbor 39
Nearest Neighbor Deconvolution 71
Nearest Neighbors 116
Neural Network 115, 116
neural network 133
Non-Linear Filters 62
NumberBelowRange 86
NumberInRange 86
NumberOfNodes 143
NumberOfObjects 104, 112
NumberOfVoxels 85

O

Object Map File Format 107
Object Maps 103
object maps 2
object of interes 138
ObjectMap 139
Oblique Planes 38
Opacity 95, 106
OpacityThickness 95, 106
open 31
Orientation 85
Orthogonal Planes 37
Ousterhout 159

P

Padding 45
PAPYRUS 24
Parzen Window 115
Parzen Windows 116
PBuffer 97
Perspective Rendering 98
PerspectiveType 98
PGM 25
photo 162, 163, 164
PIC 25
PixelsPerImage 7
Plane Extraction 37
polygonal budget 135
polygonal surface 134
PolygonBudget 141
PostScript 17
PPM 22, 24
Process 2
PSF 58

Q

query 31

R

RangeMaximum 86
RangeMinimum 86
Ray-Casting 113
read 31
Region Growing 74, 113
Render Buffer 97
Render Mask 98
Render Type 93
Render Type Algorithms 94
Render Values 98
RenderDepth 97
RenderHeigh 98
RenderHeight 97
RenderMask 95, 98
RenderWidth 97, 98
Resizin 45
Resizing 46
Resource 2
ROI 2
RotationInterval 126
RotationNutation 126
RotationPrecession 126
RotationSpin 126

S

Sagittal 37
Scattergrams 121
seed 74
Segment 2
segmentation 43
SGI 24
Shades 104
Shifting 41
ShowObject 104
SiemensCT 24
Sigma 62, 118, 120
SINGLE_VOXEL 95
Six and Twenty Six Connected Neighbors 74
SliceNumber 85
SMIS 25
Spatial Convolution 57
Spatial Filter Operations 57
spatial filters 57
Specular Reflection 99

SpecularExponent 99
SpecularFactor 99
stacked contours 148
StartBlue 105
StartGreen 105
StartRed 105
Subregionin 45
summation projection 3
Sun Raster 17
SunRaster 24, 26
surface curvature 139
Surface Matching 125
surface normals 136
SurfaceThickness 94

T

TCL 159, 160, 161, 162, 166
Tcl/Tk 159
tclsh 160
Thresholding 43, 73
thresholding 2, 138
Thresholding example 43
ThresholdMaximum 96
ThresholdMinimum 96
Thresholds 96
TIFF 31
TK 160, 161, 164, 168
Transform 2, 37
TranslationRange 126
Transverse 37
Type 93

V

Volume Coordinate System 15
Volume rendering 3
Vorini Region 134
Voroni Tessellation 134
voxel gradient shading 3
VoxelDepth 23, 85
VoxelHeight 23, 85
VoxelsPerVolume 9
VoxelWidth 23, 85
VRIO 146
VRML 145, 146
VW_FindImageMaxMin() 43
VWPutStringInfo() 7

W

Wavefront 146
Width 7
Wiener 71
Wiener Deconvolution 70
wild card file specification 26
wish 160
write 31

X

XCenter 106
XFieldOfViewAngle 98, 99
XRotation 106
XRotationIncrement 106
XTranslation 106

Y

YFieldOfViewAngle 98, 99
YRotation 106
YTable 8
YTranslation 106
YUV 25

Z

ZBuffer 97
ZCenter 106
ZRotation 106
ZTable 9
ZTranslation 106