

MetaIO

Medical Image I/O Made Simple

Abstract

Our goal was to create a simple, flexible, cross-platform image file format that supported medical image application development.

This meant that the image file format and its associated image I/O library had to support:

- 1) Image acquisition information essential to the correct processing and alignment of medical images (e.g., voxel/element size and spacing; and image position and orientation)
- 2) MSB (Most-significant-bit = Mac/Sun) and LSB (Least-significant-bit = PC) byte ordering
- 3) Arbitrary atomic pixel types (char, unsigned char, short, float, etc.)
- 4) N dimensional data
- 5) Image data stored in one file or in one file per sub-dimensional (e.g., N-1) slice
- 6) Text-based headers that are easily read and edited
- 7) Optionally storing data in a file(s) separate from the header to simplify conversion to/from other formats

MetaImage is the text-based tagged file format for medical images that resulted. We have now extended that file format to support a variety of objects that occur in medicine such as tubes (for vessels, needles, etc.), blobs (for arbitrary shaped objects), cubes, spheres, etc. The complete library is known as **MetaIO**.

The central code of MetaImage/MetaIO is quite stable. MetaImage has been in use for several years by a wide range of research at UNC, Chapel Hill. New features are occasionally added, but backward compatibility will always be maintained.

Current info on MetaObjects is available from

<http://caddlab.rad.unc.edu/technologies#metaObjects>

Table of Contents

Abstract	1
Table of Contents	2
Introduction and Installation.....	3
Obtaining MetaIO.....	3
Installing The MetaIO Package	3
MetaImage as a stand-alone package	3
MetaIO with the NLM's Insight toolkit (ITK)	4
Quick Start: Data conversion via MetaHeaders.....	4
Reading a Brick-of-Bytes	5
Reading DICOM and Other One-Slice-Per-File Data Formats	6
MetaIO Library.....	7
MetaObjects.....	8
MetaObject.....	8
Constructors.....	8
Member functions.....	8
Field descriptions.....	8
Example.....	10
MetaBlob.....	12
Example.....	12
MetaEllipse.....	13
Example.....	13
MetaGroup.....	13
MetaImage.....	13
Constructors.....	13
Member functions HeaderSize:	14
Field descriptions.....	15
Example.....	15
MetaLandmark	16
Example.....	16
MetaLine	16
Example.....	17
MetaSurface.....	17
Example.....	17
MetaTube.....	18
Example.....	18
MetaScene	18
Member functions.....	18
Example.....	19
Annex: Output File Example.....	19
Annex: Spatial Objects.....	20
Reference: Tags of MetaImage.....	20
MetaObject Tags	20
Tags Added by MetaImage.....	21

Introduction and Installation

Obtaining MetaIO

MetaIO is being distributed over the WWW by its developers in the CADDLab at UNC:

<http://caddlab.rad.unc.edu/technologies#metaObjects>

MetaIO is also being distributed with the National Library of Medicine's Insight Toolkit (ITK) for medical image segmentation and registration:

<http://www.itk.org>

Installing The MetaIO Package

MetaIO is a hierarchy of C++ classes and functions. We have yet to find a modern C++ compiler that does not compile MetaIO. Known compatible compilers include G++ v2.95 and beyond (and probably previous), Microsoft Visual C++ 6.0, Sun's CC on Solaris 2.6 and beyond, Intel compiler and compilers on other workstations including HPs, SGIs, and Alpha systems. Please contact us (Stephen R. Aylward, aylward@unc.edu, Julien Jomier jomier@unc.edu or via <http://caddlab.rad.unc.edu>) if you encounter any incompatibilities between our code and your compiler.

The hierarchy of the software in the stand-alone MetaIO package is as follows:

MetaIO/

doc/

tests/

The top level contains the source files, the header files, and the CMakeLists.txt file that is used by the CMake program to compile MetaIO. This document and the MetaObjects www pages are in the doc directory. A sequence of simple tests is available in the tests directory.

The hierarchy of the software in the ITK distribution is as follows:

Insight/Code/Utilities/MetaIO/

doc/

tests/

Insight/Code/IO/

Insight/Examples/

MetaImageReadWrite/

MetaImageViewer/

MetaImageColorViewer/

MetaImageImporter/

The files in Insight/Code/Utilities exactly match those in the stand-alone MetaIO package. Routines to wrap metaImage so that it can be accessed via the itkImageIO Object Factory are in /Insight/Code/IO. The details of the Examples are provided later in this documents.

MetaImage as a stand-alone package

If you downloaded MetaIO separate from Insight, you must also download and install CMake from www.CMake.org. That www site provide details regarding installing CMake and using it to compile another program. Follow those directions to create the MetaIO library and tests.

A minimal description of the process is presented next.

- 1) Install CMake
- 2) On Windows:
 - a. Run CMake.exe
 - b. In the source directory field, browse to the top-level of the MetaIO directory
 - c. In the binary directory field, enter the path and name of the source directory, but add “-VC++” to the directory name (this assumes that you are compiling using VC++, otherwise add an equally descriptive name for the directory to store the binaries in such as “MetaIO-bcc” for Borland).
 - d. Press the “configure” button. It will ask if you want to create the binary directory, press “yes.”
 - e. Specify your compiler in the pull-down menu field.
 - f. Press the “configure” button a second time, and then press “okay” to generate the Makefiles for your compiler (e.g., MetaIO.dsw and related files for VC++) in the binary directory.
- 3) On *nix/Cygwin machines:
 - a. At an appropriate spot, create a directory “MetaIO-g++” or give it an equally descriptive name based on your compiler
 - b. “cd MetaIO-g++”
 - c. Run “cmake <path to MetaIO source directory>”
 - d. Press “g” to generate the makefiles for your compiler in the binary directory.

To use MetaIO in your stand-alone programs:

- 1) add the MetaIO directory to your include paths,
- 2) add “#include <metaImage.h>” to the top of the your file that performs your image IO,
- 3) add the MetaIO binary directory to your link paths, and
- 4) link with the MetaIO library.

See the files in MetaIO/tests for examples of how to read/write MetaImages and other MetaObjects.

MetaIO with the NLM’s Insight toolkit (ITK)

If you have downloaded and are installing MetaIO as part of the Insight toolkit, follow the standard installation procedure of Insight. This will create the MetaIO library and the ITK specific wrapping, examples and tests. Certain examples, such as the MetaImageViewer, also require FLTK (a cross-platform user interface library available from <http://fltk.org>). Install FLTK and then ITK to have every MetaIO example built. Numerous other ITK examples also rely on FLTK.

See the file /Insight/Examples/MetaImageReadWrite for a working example on how to develop a program using MetaImage for IO. Work is underway to add access to other MetaObjects (e.g., tubes, spheres, etc.) via ITK’s SpatialObjects in /Insight/Code/SpatialObject.

Quick Start: Data conversion via MetaHeaders

This section assumes that you have some data that you wish to process using an application that reads MetaImages. This section gives examples on how “convert” your data to the MetaImage format.

For uncompressed data (i.e., data stored in a raw format as is often done for DICOM, BMP, and PNG formats), “conversion” to MetaImage is actually just a matter of specifying a MetaImage Headerfile (a “MetaHeader”) that describes and points to the file(s) containing your data.

For compressed data (i.e., data stored in JPEG or GIF formats), you must first convert your data to a non-compressed format. Currently, no data compression methods are supported by MetaImage, but adding support for .gz, RLE, and other compression formats is high on our To-Do list. One of the most robust

image conversion software packages is ImageMagick (<http://www.imagemagick.org/>; Unix and PC versions available). It has an application called “convert” that handles most of the popular 2D image formats.

IF YOU HAVE INSTALLED METAIO AS PART OF ITK, USE THE PROGRAM Insight/Examples/MetaImageImporter. IT WILL ASK A SERIES OF QUESTIONS AND PRODUCE A METAIMAGE HEADER FOR YOU.

Reading a Brick-of-Bytes

A “brick of bytes” is a volume of image data stored in a single file possibly with preceding and trailing non-image data.

To correctly load these images, the minimal information that you need to know is:

- 1) Number of dimensions
- 2) Size of each dimension
- 3) Data type
- 4) Name of the data file

For example, let’s say the data was 3 dimensional, had 256 x 256 x 64 voxels, used an unsigned short to represent the value at each voxel, and was stored in the file “image.raw”. The resulting MetaHeader (our naming convention would call this file “image.mhd”) file would read

```
ObjectType = Image
NDims = 3
DimSize = 256 256 64
ElementType = MET_USHORT
ElementDataFile = image.raw    (this tag must be last in a MetaImageHeader)
```

That’s it, but this assumes quite a bit about the image data. Specifically, it assumes

- 1) There are not any non-image data bytes (header data) at the beginning of the image data file “image.raw”.
- 2) The voxels are cubes – the distance spanned by and between a voxel in each coordinate direction is 1 “unit”, e.g., 1x1x1mm voxel size and voxel spacing
- 3) The byte-order of the data in image.raw matches the byte ordering native to the machine the application is running on (e.g., PC’s use LSB ordering and Suns/Macs use MSB ordering).

If these assumptions are false, the data will not be loaded correctly by the application. To fix these problems....

- 1) To skip the header bytes in the image data file, use
HeaderSize = X
where X is the number of bytes to skip at the beginning of the file before reading image data. If you know there are no trailing bytes (extra bytes at the end of the file) you can specify
HeaderSize = -1
and MetaImage will automatically calculate the number of extract bytes in the data file, assume they those bytes are at the head of the data file, and automatically skip them before beginning to read the image data.
- 2) To specify the spacing of the voxels, use
ElementSpacing = X Y Z
where X is the distance between of the centers of the voxels along the x-dimension, Y is the spacing in the y-dimension, and Z is the spacing in the z-dimension. Therefore, to specify a 1x1x3mm voxel spacing, use
ElementSpacing = 1 1 3

NOTE: If ElementSpacing is not specified, it is assumed to be equal to ElementSize. If neither is specified, both are assumed to be 1.

- 3) To specify a voxel size, use
ElementSize = X Y Z
where X Y Z represent the size in the x, y, and z-dimensions respectively.

NOTE: If ElementSize is not specified, it is assumed to be equal to ElementSpacing. If neither is specified, both are assumed to be 1.

- 4) To specify a particular byte ordering, use
ElementByteOrderMSB = True
or
ElementByteOrderMSB = False
MSB (aka big-endian) ordering is common to SPARC and Motorola processors (e.g., Macintoshes). LSB (aka little-endian) ordering is common to Intel processors (e.g., PC compatibles).

Putting it all together, to “convert” a file containing the image data in a continuous block at the end of the file, specify the header

```
ObjectType = Image
NDims = 3
DimSize = 256 256 64
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = image.raw
```

Reading DICOM and Other One-Slice-Per-File Data Formats

If the data is split to be one slice per file, as is done with DICOM data, only the ElementDataFile tag's option needs to change.

Since the MetaLibrary cannot automatically parse DICOM headers, those headers must be skipped and the user must specify the image dimensions and other essential image information. For DICOM files, the MetaLibrary must automatically calculate the header size of each file (luckily for almost every type of DICOM object in common use, the image data is stored at the end).

To specify which files comprise the volume, they can be specified as an ordered list in the MetaHeader using the ElementDataFile=LIST option. The filenames should be listed at the end of the MetaHeader, after the ElementDataFile option, and the filenames should be separated by whitespace:

```
ObjectType = Image
NDims = 3
DimSize = 512 512 100
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = LIST
filenameOfSlice1
```

```
filenameOfSlice2
filenameOfSlice3
filenameOfSlice4
```

. (one hundred filenames must be specified to specify the 100 slices in the volume)

The second way of specifying a series of files can be used if the filenames are numerically distinguished. That is, the files names should be able to be specified using a numeric substitution into a c-style printf-string, for a range of values. In pseudo-code:

```
for i=numBegin to numEnd step numStep
    sprintf(sliceName, "baseName.%03d", i);
end
```

The parameters of this system are numBegin, numEnd, numStep, and the c-style printf string (e.g., "baseName.%03d"). The begin, end, and step parameters appear in order after the c-style printf string:

```
ObjectType = Image
NDims = 3
DimSize = 512 512 100
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = baseName.%03d 1 100 1
```

This MetaImage will cause the files "baseName.001" to "baseName.100" to be read to create a 100-slice volume.

In this case, because of the overlap of the slices, it may be helpful to only consider every-other slice in the volume. Changing the slice spacing and the ElementDataFileNumStep enacts this...

```
ObjectType = Image
NDims = 3
DimSize = 512 512 50
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 2
ElementByteOrderMSB = False
ElementDataFile = baseName.%03d 1 100 2
```

The complete set of MetaImage Tags are given in the Reference section of this document. The next section discusses how to use the MetaImage Library for image reading and writing in your own programs.

MetaIO Library

The base class of the MetaIO library is the MetaObject class. It defines a base set of tags that are common to all metaObjects such as MetaImages, MetaTubes, etc.

The tags are defined using the protected member functions SetupReadFields and SetupWriteFields. These functions create a list of MetaFieldRecords to define the name, type, interdependence, and necessity of each tag. Helper functions for defining the fields are in MetaUtils.cxx. The types are defined via enums in MetaTypes.h

The derived classes add tags to the list via their own SetupReadFields and SetupWriteFields member functions. The MetaImage subclass also re-implements the Read and Write methods since non tag data (i.e., the pixel values) must also be read. Compare the derived classes for MetaCube and MetaImage.

MetaObjects

In this section we describe the metaObjects which have been implemented already. If you want to implement other objects, you can easily derive these classes. metaObject is the base class for metaIO. metaScene and metaGroup are also a useful objects that support multiple metaObjects. All these objects are described in details next.

MetaObject

Constructors

Simple constructor

```
MetaObject(void);
```

Read a metafile and store the result in the current object

```
MetaObject(const char * _fileName);
```

Define the dimension of the object at construction time.

```
MetaObject(unsigned int dim);
```

Member functions

Specify the filename to read (Optional)

```
void FileName(const char * _fileName);  
const char * FileName(void) const;
```

Read a MetaFile

```
bool Read(const char * _fileName=NULL);
```

Write a MetaFile

```
bool Write(const char * _fileName=NULL);  
virtual bool Append(const char * _headName=NULL);
```

Print the info about the metaObject

```
virtual void PrintInfo(void) const;
```

Clear the information as well as the data of the metObject

```
virtual void Clear(void);
```

Field descriptions

Name:

```
void Name(const char * _Name);  
const char * Name(void) const;
```

Color:

```
const float * Color(void) const;  
void Color(float _r, float _g, float _b, float _a);  
void Color(const float * _color);
```


ID:*ID number of the current metaObject*

```
void ID(int _id);
int ID(void) const;
```

Parent ID:*ID number of the parent metaObject*

```
void ParentID(int _parentId);
int ParentID(void) const;
```

Binary Data:*Specify if the data is binary or not*

```
void BinaryData(bool _binaryData);
bool BinaryData(void) const;
```

Binary Data Byte Order:

```
void BinaryDataByteOrderMSB(bool _binaryDataByteOrderMSB);
bool BinaryDataByteOrderMSB(void) const;
```

Comments:

```
const char * Comment(void) const;
void Comment(const char * _comment);
```

Object Typename and optional subtype (i.e. the type of the object)

```
const char * ObjectTypeName(void) const;
void ObjectTypeName(const char * _objectTypeName);
const char * ObjectSubTypeName(void) const;
void ObjectSubTypeName(const char * _objectSubTypeName);
```

Associated transformations:

Physical location (in millimeters and with respect to machine coordinate system or the patient) of the first element in the image. Physical orientation of the object is defined as an NDims x NDims matrix.

Offset:

```
const float * Offset(void) const;
float Offset(int _i) const;
void Offset(const float * _position);
void Offset(int _i, float _value);
```

Position:

```
const float * Position(void) const;
float Position(int _i) const;
void Position(const float * _position);
void Position(int _i, float _value);
```

Origin:

```
const float * Origin(void) const;
float Origin(int _i) const;
void Origin(const float * _position);
void Origin(int _i, float _value);
```

Rotation:

```
const float * Rotation(void) const;
```

```
float Rotation(int _i, int _j) const;
void Rotation(const float * _orientation);
void Rotation(int _i, int _j, float _value);
```

Orientation:

```
const float * Orientation(void) const;
float Orientation(int _i, int _j) const;
void Orientation(const float * _orientation);
void Orientation(int _i, int _j, float _value);
```

Center of rotation of the object:

```
const float * CenterOfRotation(void) const;
float CenterOfRotation(int _i) const;
void CenterOfRotation(const float * _position);
void CenterOfRotation(int _i, float _value);
```

Anatomical Orientation:

```
const char * AnatomicalOrientationAcronym(void) const;
const MET_OrientationEnumType * AnatomicalOrientation(void) const;
MET_OrientationEnumType AnatomicalOrientation(int _dim) const;
void AnatomicalOrientation(const char * _ao);
void AnatomicalOrientation(const MET_OrientationEnumType * _ao);
void AnatomicalOrientation(int _dim, MET_OrientationEnumType _ao);
void AnatomicalOrientation(int _dim, char ao);
```

Element Spacing:

Physical Spacing (in same units as position)

```
const float * ElementSpacing(void) const;
float ElementSpacing(int _i) const;
void ElementSpacing(const float * _elementSpacing);
void ElementSpacing(int _i, float _value);
```

For simplicity, some dynamic functions have been recently added. They allow the user to add fields dynamically.

The function AddUserField is defined by:

```
template <class T>
bool AddUserField(const char* _fieldName, MET_ValueEnumType _type, int _length,
                 T * _v, bool _required=true, int _dependsOn=-1 )
```

The user may also want to clear the fields created by using ClearUserFields().

```
void* GetUserField(const char* _name);
```

Note: When using GetUserField() function, the user is responsible for the deletion of the pointer created. See the following example for details.

Example

```
/** We create a simple 3D metaObject with some properties */
```

```

MetaObject tObj(3); // Create a 3D metaObject
tObj.FileName("testObject.txt"); // Define the name of the file
tObj.Comment("TestObject"); // Add some comments
tObj.ObjectTypeName("Object"); // Define the type of the object
tObj.ObjectSubTypeName("MinorObject"); // and the subtype as well

/** We now define the position and the orientation as well as the spacing of the created object */
// The position part
tObj.Position(0, 1);
tObj.Position(1, 2);
tObj.Position(2, 3);

// The orientation part
float orient[9];
int i;
for(i=0; i<9; i++)
{
    orient[i] = 0;
}
orient[0] = 1;
orient[5] = 1;
orient[7] = 1;
tObj.Orientation(orient);

// The element spacing part
tObj.ElementSpacing(0, 1);
tObj.ElementSpacing(1, 2);
tObj.ElementSpacing(2, 1);

/** Add user's defined fields */
tObj.AddUserField("MyName", MET_STRING, strlen("JulienAndStephen"), "JulienAndStephen");

/** Write the object */
tObj.Write();

/** Clear completely the object */
tObj.Clear();
tObj.ClearUserFields();

/** Specify that we want to read the field 'MyName' */
tObj.AddUserField("MyName", MET_STRING);

/** Read the object */
tObj.Read("testObject.txt");

/** Print the object */
tObj.PrintInfo();

/** Get the name in the file */
char* name = static_cast<char*>(tObj.GetUserField("MyName"));
std::cout << name << std::endl;

/** delete the allocated pointer */
delete [] name;

```

All of the following objects derive from `metaObject`.

MetaBlob

A blob is defined by a list of points that describe the object. The points can be inside the object (if obtained by connected-component for instance) or only on the surface. Note that a color (RGBA) can be associated with each point.

The required fields are:

The number of points defining the object:

`NPoints(int npnt);`

How the position of the points is stored in the file. By default the configuration is x y z red green blue alpha

`PointDim(const char* pointDim);`

To access the internal list of points user should use the `GetPoints()` function which returns the internal list by reference. Note that the list is a list of pointers to point and is deleted automatically by the object itself so the user does not need to free the allocated memory.

Example

```
/** Create a 3D blob */
MetaBlob blob(3);
blob.ID(0); // define the ID of the blob

/** Add 10 points to the blob */
BlobPnt* pnt;

unsigned int i;
for(i=0;i<10;i++)
{
    pnt = new BlobPnt(3);
    pnt->m_X[0]=(float)0.2;
    pnt->m_X[1]=i;
    pnt->m_X[2]=i;
    blob.GetPoints().push_back(pnt); // push the created point into the list of points
}

/** Write the blob in binary format */
blob.BinaryData(true);
blob.ElementType(MET_FLOAT);
blob.Write("myBlob.meta");

/** Read the file */
blob.Read("myBlob.meta");
blob.PrintInfo();

/** Access the list of points */
std::cout << "Accessing pointlist..." << std::endl;

MetaBlob::PointListType plist = blob.GetPoints();
MetaBlob::PointListType::const_iterator it = plist.begin();
```

```

while(it != plist.end())
{
    for(unsigned int d = 0; d < 3; d++)
    {
        std::cout << (*it)->m_X[d] << " ";
    }

    std::cout << std::endl;
    it++;
}

```

MetaEllipse

MetaEllipse is an N-Dimensional object to define ellipsoids like circles, spheres or even hyper-ellipsoids. The only field you need to provide is the Radius.

There are several ways to input the radius:

- a) As an array of floats: void Radius(const float* radius);
- b) As a single value which means that we are defining an hyper-sphere: void Radius(float radius);
- c) A convenient way to define a 2D ellipse: void Radius(float r1,float r2);
- d) A convenient way to define a 3D ellipse: void Radius(float r1,float r2, float r3);

Example

```

/** Create a sphere */
MetaEllipse myEllipse (3);
myEllipse ->Radius(3); // radius of 3

```

MetaGroup

MetaGroup does not have added functionalities compared to metaObject. It allows to group object in a metafile.

MetaImage

Constructors

Simple constructor by specifying the filename

```
MetaImage(const char *_headerName);
```

Constructor by shared memory

```
MetaImage(MetaImage *_im);
```

Other constructors

```

MetaImage(int _nDims,
           const int *_dimSize,
           const float *_elementSpacing,
           MET_ValueEnumType _elementType,

```

```

    int _elementNumberOfChannels=1,
    void *_elementData=NULL);

MetaImage(int _x, int _y,
    float _elementSpacingX,
    float _elementSpacingY,
    MET_ValueEnumType _elementType,
    int _elementNumberOfChannels=1,
    void *_elementData=NULL);

MetaImage(int _x, int _y, int _z,
    float _elementSpacingX,
    float _elementSpacingY,
    float _elementSpacingZ,
    MET_ValueEnumType _elementType,
    int _elementNumberOfChannels=1,
    void *_elementData=NULL);

```

Member functions

HeaderSize:

Return the size of the header.

```
int HeaderSize(void) const;
```

Quantity:

Total number of elements in the image.

```
int Quantity(void) const;
```

SubQuantity:

Number of elements in image spanning sub-dimensions. E.g., elements per line, 2D sub-image, 3D sub-volume.

```
const int * SubQuantity(void) const;
int SubQuantity(int _i) const;
```

ElementMin/Max:

The default max returned is the largest allowed by ElemNBytes (12 bit uint16_t will give 4096 max). This may not represent the true max. Use _reCalc=true to force a calculation of the actual max element value.

```
bool ElementMinMaxValid(void) const;
void ElementMinMaxValid(bool _elementMinMaxValid);
void ElementMinMaxRecalc(void);
double ElementMin(void) const;
void ElementMin(double _elementMin);
double ElementMax(void) const;
void ElementMax(double _elementMax);
```

ElementByteOrderSwap:

These functions are available only after ReadImageData() or if _read_and_close=TRUE when read

```
void ElementByteOrderSwap(void);
bool ElementByteOrderFix(void);
```

ConvertTo:

Converts to a new data type. Rescales using Min and Max.

```
bool ConvertElementDataTo(MET_ValueEnumType _elementType=MET_UCHAR,
    double _toMin=0, double _toMax=0);
```

Field descriptions

Modality:

Specify the modality of the image

```
MET_ImageModalityEnumType Modality(void) const;  
void Modality(MET_ImageModalityEnumType _modality);
```

Dimension size:

Specify the size of the image in each dimension

```
void DimSize(const int * _dimSize);  
void DimSize(int _i, int _value);
```

SequenceID:

DICOM designation of this image relative to other images acquired at the same time

```
const float * SequenceID(void) const;  
float SequenceID(int _i) const;  
void SequenceID(const float * _sequenceID);  
void SequenceID(int _i, float _value);
```

ElementSize:

Optional Field. Physical size (in MM) of each element in the image (0 = xSize, 1 = ySize, 2 = zSize)

```
const float * ElementSize(void) const;  
float ElementSize(int i) const;  
void ElementSize(const float * _pointSize);  
void ElementSize(int _i, float _value);
```

ElementType:

Pixel type

```
MET_ValueEnumType ElementType(void) const;  
void ElementType(MET_ValueEnumType _elementType);
```

ElementNumberOfChannels:

Number of channels

```
int ElementNumberOfChannels(void) const;  
void ElementNumberOfChannels(int _elementNumberOfChannels);
```

ElementData:

Returns a pointer to the data.

```
void * ElementData(void);  
double ElementData(int _i) const;  
void ElementData(void * _data);  
bool ElementData(int _i, double _v);
```

ElementDataFileName:

Set/Get the filename

```
const char * ElementDataFileName(void) const;  
void ElementDataFileName(const char * _dataFileName);
```

Example

```
ObjectType = Image  
NDims = 2  
BinaryData = True
```

```

BinaryDataByteOrderMSB = False
ElementSpacing = 1 2
DimSize = 8 8
ElementType = MET_CHAR
ElementDataFile = LOCAL
[Pixel Data]

```

MetaLandmark

MetaLandmark is a simple list of landmarks.

The number of landmarks defining the object is set using the function

NPoints(int npnt);

How the position of the points is stored in the file. By default the configuration is x y z red green blue alpha

PointDim(const char* pointDim);

To access the internal list of points user should use the GetPoints() function which returns the internal list by reference. Note that the list is a list of pointers to point and is deleted automatically by the object itself so the user does not need to free the allocated memory.

Example

```

/** Create a 3D Landmark */
MetaLandmark Landmark(3);
Landmark.ID(0);
LandmarkPnt* pnt;

/** Add some landmarks to the list of landmark points*/
for(unsigned int i=0;i<10;i++)
{
    pnt = new LandmarkPnt(3);
    pnt->m_X[0]=(float)0.2;
    pnt->m_X[1]=i;
    pnt->m_X[2]=i;
    Landmark.GetPoints().push_back(pnt);
}

```

MetaLine

A metaLine is actually a polyline defined by a list of connected points.

A point on the line has a given position, a normal and a color.

To set the position the local variable m_X should be filled in the point structure. The variable m_V which is a double pointer to a float is used to assess the normal. The normal has the dimension of the object minus one since a metaLine in a 3D space will have two normals (a plane).

Note that the user does not need to allocate the memory for those variables, this is done automatically in the constructor of the point.

To access the internal list of points user should use the GetPoints() function which returns the internal list

by reference. Note that the list is a list of pointers to point and is deleted automatically by the object itself so the user does not need to free the allocated memory.

Example

```
/** Create a 3D MetaLine */
MetaLine Line(3);
LinePnt* pnt;

for(unsigned int i=0;i<10;i++)
{
    pnt = new LinePnt(3);

    /** Define the position */
    pnt->m_X[0]=(float)0.2;
    pnt->m_X[1]=i;
    pnt->m_X[2]=i;

    /** Define the normals */
    pnt->m_V[0][0]=(float)0.3;
    pnt->m_V[0][1]=i;
    pnt->m_V[0][2]=i;
    pnt->m_V[1][0]=(float)0.4;
    pnt->m_V[1][1]=i+1;
    pnt->m_V[1][2]=i+1;
    Line->GetPoints().push_back(pnt);
}

/** Write the result */
Line.BinaryData(true);
Line.Write("myLine.meta");
```

MetaSurface

The definition of a metaSurface is quite similar to the metaLine's, except for the normal which is only a NDim vector (i.e. an array of floats) where NDim is the dimension of the metaObject.

To access the internal list of points user should use the GetPoints() function which returns the internal list by reference. Note that the list is a list of pointers to point and is deleted automatically by the object itself so the user does not need to free the allocated memory.

Example

```
MetaSurface surface(3);
SurfacePnt* pnt;

for(unsigned int i=0;i<10;i++)
{
    pnt = new SurfacePnt(3);

    /** Position */
    pnt->m_X[0]=(float)0.2;
    pnt->m_X[1]=i;
    pnt->m_X[2]=i;
```

```

/* Normal */
pnt->m_V[0]=(float)0.8;
pnt->m_V[1]=i;
pnt->m_V[2]=i;
surface->GetPoints().push_back(pnt);
}

```

MetaTube

A metaTube is a tubular structure defined by a list of connected points (like a metaLine) but more fields have been added for a complete representation, especially the one of blood vessels.

To specify a point that belongs to the tube, the user can define: the position, the radius at that point, the normal(s), the tangent, the color, the Identification number, the medialness, the branchness, the ridgeness, and three alpha values that represents the ratio of the eigen values at that points.

Note that metaTube supports only 2D and 3D tubes.

Also for a metaTube, the ID of the root can be specified by the command `Root(int rootID)` and the ID of the parent point can also be assessed using `ParentPoint(int parentpoint)`.

To access the internal list of points user should use the `GetPoints()` function which returns the internal list by reference. Note that the list is a list of pointers to point and is deleted automatically by the object itself so the user does not need to free the allocated memory.

Example

```

/** Create a 3D tube*/
MetaTube* tube1 = new MetaTube(3);
tube1->ID(0);

/** Add 10 points to the list of tubePoints */
TubePnt* pnt;
for(unsigned int i=0;i<10;i++)
{
    pnt = new TubePnt(3);

    pnt->m_X[0]=i; // position
    pnt->m_X[1]=i;
    pnt->m_X[2]=i;
    pnt->m_R=i; // radius
    tube1->GetPoints().push_back(pnt);
}

```

MetaScene

A metaScene is a metaObject that contains a flat list of metaObjects.

Member functions

Add an object to the scene:

```
void AddObject(MetaObject* object);
```

Return the number of objects in the scene:

```
int NObjects(void) const;
```

Get a list of objects present in the scene:

```
ObjectListType * GetObjectList(void) {return & m_ObjectList;}
```

Example

```
/** Define a 3D Scene */
MetaScene scene(3);

MetaEllipse * e1 = new MetaEllipse(3);
e1->ID(0);
e1->Radius(3);

MetaGroup g0;
MetaGroup * g1 = new MetaGroup(3);
g1->ID(2);

s->AddObject(g1);
s->AddObject(e1);

s->Write("scene.scn");
scene.Clear();

s->Read("scene.scn");
```

Annex: Output File Example

Here is the example of a metafile with a scene that contains metaObjects

```
ObjectType = Scene
NDims = 3
NObjects = 3
ObjectType = Group
NDims = 3
ID = 2
EndGroup =
ObjectType = Ellipse
NDims = 3
ID = 0
ParentID = 2
Radius = 1 2 3
ObjectType = Line
NDims = 3
ID = 0
BinaryData = False
BinaryDataByteOrderMSB = False
ElementType = MET_FLOAT
PointDim = x y z v1x v1y v1z
NPoints = 3
Points =
1 2 3 0 0 0
1 2 3 0 0 0
```

```

1 2 3 0 0 0
ObjectType = Landmark
NDims = 3
ID = 0
BinaryData = True
BinaryDataByteOrderMSB = False
ElementType = MET_FLOAT
PointDim = x y z red green blue alpha
NPoints = 2
Points =
1 2 3 1.0 0.0 0.0 1.0
1 2 3 1.0 0.0 0.0 1.0
1 2 3 1.0 0.0 0.0 1.0

```

Annex: Spatial Objects

MetaIO has also been chosen to support Spatial Objects IO. To obtain a complete documentation of Spatial Objects and how to read/write them out please see the Insight user's manual available at www.itk.org.

Reference: Tags of MetaImage

MetaObject Tags

The tags of MetaObject are:

Comment	– MET_STRING – User defined - arbitrary
ObjectType	– MET_STRING – Defined by derived objects – e.g., Tube, Image
ObjectSubType	– MET_STRING – Defined by derived objects – currently not used
TransformType	– MET_STRING – Defined by derived objects – e.g., Rigid
NDims	– MET_INT – Defined at object instantiation
Name	– MET_STRING – User defined
ID	– MET_INT – User defined else -1
ParentID	– MET_INT – User defined else -1
BinaryData	– MET_STRING – Are the data associated with this object stored at Binary or ASCII – Defined by derived objects
ElementByteOrderMSB	– MET_STRING
BinaryDataByteOrderMSB	– MET_STRING
Color	– MET_FLOAT_ARRAY[4] – R, G, B, alpha (opacity)

Position	– MET_FLOAT_ARRAY[NDims] – X, Y, Z,... of real-world coordinate of 0,0,0 index of image)	
Orientation	– MET_FLOAT_MATRIX[NDims][NDims]	– [0][0],[0][1],[0][2] specify X, Y, Z... direction in real-world of X-axis of image – [1][0],[1][1],[1][2] specify X, Y, Z... direction in real-world of Y-axis of image, etc.
AnatomicalOrientation	– MET_STRING – Specify anatomic ordering of axis. For example, RAS mean x-axis is Right to Left, y-axis is Anterior to Posterior, and Z-axis is Superior to Inferior. Use only [R L] [A P] [S I] per axis.	
ElementSpacing	– MET_FLOAT_ARRAY[NDims] – The distance between voxel centers	

Tags Added by MetaImage

In addition to the above tags, MetaImage provides the following tags:

DimSize	– MET_INT_ARRAY[NDims] – Number of elements per axis in data	
HeaderSize	– MET_INT	– Number of Bytes to skip at the head of each data file. Specify –1 to have MetaImage calculate the header size based on the assumption that the data occurs at the end of the file.
Modality	– MET_STRING – One of enum type: MET_MOD_CT, MET_MOD_MR, MET_MOD_US... See MetaImageTypes.h	
SequenceID	– MET_INT_ARRAY[4] – Four values comprising a DICOM sequence: Study, Series, Image numbers	
ElementMin	– MET_FLOAT – Minimum value in the data	
ElementMax	– MET_FLOAT – Maximum value in the data	
ElementNumberOfChannels	– MET_INT – Number of values (of type ElementType) per voxel	
ElementSize	– MET_FLOAT_ARRAY[NDims] – Physical size of each voxel	
ElementType	– MET_STRING – One of enum type: MET_UCHAR, MET_CHAR... See MetaImageTypes.h	
ElementDataFile	– MET_STRING – One of the following: <ul style="list-style-type: none"> - Name of the file to be loaded - A printf-style string followed by the min, max, and step values to be used to pass an argument to the string to create list of file names to be loaded (must be (N-1)D blocks of data per file). - LIST [X] – This specifies that starting on the next line is a list of files (one filename per line) in which the data is stored. Each file (by default) contains an (N-1)D block of data. If a second argument is given, its first character must be a number that specifies the dimension of the data in each file. For example ElementDataFile = LIST 2D means that there will be a 2D block of data per file. - LOCAL – Indicates that the data begins at the beginning of the next line. 	

