Comprehensive C++ I/O Libraries Supporting Image Processing in a University Research Environment

Alfred Graham Gash
University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, North Carolina 27599–3175

Fred J. R. Appelman and Karel J. Zuiderveld
3D Computer Vision Group, University of Utrecht, Heidelberglaan 100, 3584 CX Utrecht, The Netherlands

ABSTRACT

An on-going effort to develop a set of class-based libraries in C++ for image I/O and image processing is described. These new libraries will provide the following features to application programs based on them:

Derivable image classes
Machine-architecture independence of data storage
Fast, user-customizable image I/O
Unlimited number of descriptive data items
Arbitrary number of image dimensions
User-definable data types
Optional automatic type conversion and compression
Support for accessing images using reference files and pipes

This highly-flexible and powerful capability is designed to meet the needs of workstation-based image processing and computer vision research in a multi-departmental university environment.

1. HISTORICAL BACKGROUND

For more than two decades, the Computer Science department of the University of North Carolina has been doing research in the areas of image processing and computer vision, with primary emphasis on medical imaging applications of these techniques. This work has often been done in collaboration with physicians in the N.C. Memorial Hospital. During this time, a large body of software based on the UNIX† operating system has been written to support this research. Recently, a strong collaboration has been established with various departments at the University of Utrecht in the Netherlands, and the 3D Computer Vision (3DCV) group there has become a co-developer of this software.

The image processing system which has resulted is called /usr/image[1]. It consists of a central library (archived set of compiled functions) for image file access (input and output) and a suite of programs, which call functions in the library. The I/O library (called libim.a) originated in 1981. It supports a simple file format for storing pixels and a few related fields. A primitive mechanism for extending the format to contain other kinds of data is supported but has rarely been used.

The old format is not directly visible to users. Instead, it is encapsulated in the set of functions in the linkable library. There are presently 23 such functions, with an additional 9 for storing and retrieving physical dimensioning data on clinical patients.

2. MOTIVATION

The requirements of modern image processing and computer vision on today's high-speed uniprocessors, especially workstations, and the object-oriented techniques now available using languages such as C++ have

† UNIX is a registered trademark of AT&T.

made the old format and library obsolete. In addition, we have encountered a number of practical problems that increase the time of turning ideas into working programs.

Among them is the fact that some machines do not store integer data in the same byte order as other machines. Sun Microsystems has proposed[2] a widely accepted mechanism called "External Data Representation," or XDR, to overcome this problem.

At about the same time, the ACR-NEMA standard for "Digital Imaging and Communications" was published.[3] This standard is important, because it specifies a common format for all data available from medical imaging devices. Our old format could not easily be extended to store the many data fields available from such devices. This has been a problem as we have tried to apply our software to real clinical situations.

Other problems are evident in this software. For example, there is no compression capability, so obtaining adequate disk space has often been a problem. There is no run-time method for converting pixels from one type to another. Some in our group want to store the pixel data separately from the descriptive data, while the format supports only a single, combined type of file. And so forth.

Various groups have produced significant image processing and analysis packages in recent years. Well-known examples are the Khoros package[4] and the Mayo Foundation's ANALYZE[5]. There is also at least one general purpose format for scientific applications, the "Hierarchical Data Format" (HDF)[6] available from the National Center for Supercomputing Applications (NCSA).

Nevertheless, we have not found any one package that is sufficient for the needs of our research, for a variety of reasons. Often, this is because the package does not provide a comprehensive solution to the problems described above and to the needs of our highly diverse environment.

3. THE ENVIRONMENT

The collaborative groups at the University of North Carolina and the University of Utrecht represent a total of 18 academic departments. There are multiple groups working on disparate projects having various image processing needs. The common theme that relates them is the use of modern workstations, on which they run either the /usr/image package or a related C++ package[7] that uses the same format. There are relatively few situations where special image-processing (e.g., DSP) boards are used.

A collaborative exchange of images and software is a common feature of this environment. Thus, computer scientists and physicians often work as an inter-departmental team. The concerns of the various groups, while related, often are not identical. The hospital staff is concerned about issues of data management — confidentiality, quality control, the use of standards such as the ACR-NEMA standard, etc. In contrast, the computer scientists are most often concerned with issues of methodology and performance — algorithms, speed, display parameters, etc.

Finally, there is a need to support other disciplines because our research efforts are not all in medical imaging. For example, one UNC computer scientist collaborates with astronomers in the Department of Physics. He needs to be able to transfer FITS[8] images into the /usr/image format for processing.

Existing image file formats and their supporting libraries cannot adequately handle such a wide diversity of needs. Because of this, we undertook to build a software package that would run efficiently on modern workstations yet provide maximal capability and flexibility to the user.

4. THE DEVELOPMENT GOALS

The old software had a few features that we considered important and worth retaining in some form. Pixel access in the old library was fast and used direct access to the files. Indexing of pixels in multi-

dimensional images used the natural choice of row-major ordering imposed by the C programming language in which the libim.a library was written. There was a fairly wide, although not comprehensive, list of pixel types. However, as mentioned above, there were a number of serious problems because of the age of the old design. This section will discuss other features desired of the new software, which are intended to solve the problems described above and to provide novel capabilities.

As mentioned earlier, it was important in our collaborative environment to be able to exchange files between a variety of computer architectures. Yet since different vendors use different arithmetic encoding schemes, byte-order dependence, especially in the common case of 16-bit integer images, is frequently a problem. To achieve complete machine independence of images, we decided partially to adopt the "External Data Representation" (XDR)[2].

The use of XDR introduces a computational overhead, which can be quite large, especially with the number of pixels† in today's 3D medical images. We therefore decided only to use XDR when necessary and to put this partially under control of the user. In particular, since the header information is generally small compared to the pixel data, we elected always to use XDR for the header information and to allow the user to specify when it is used for pixels. For example, in the case of integer data, it is normally not necessary to use XDR since an in-place swapping of bytes is often more efficient. This also usually results in a space savings; for example, XDR encodes 16-bit integers into 32-bit quantities.

There are basically two ways one can store an image on disk. Either it is stored as a single file, or the pixels are put in one file with the rest of the information put into another file. Various people in our and other laboratories have advocated one or the other of these schemes for the following reasons.

Updating a combined format in-place can be computationally expensive, because of the need to move data within the file. If it is necessary to copy the file to modify it, more disk space is required. Nevertheless, all information pertaining to an image is present in a combined file, so there is no risk of losing just the pixels or the header. Moreover, users having only small images may see no reason for separated files, claiming they clutter their workspaces.

On the other hand, the use of separated files has been advocated, on the basis that the pixel file is easily mapped to special-purpose hardware, particularly display systems. With the recent introduction of optical "jukebox" systems, it is also possible to store headers on primary disk and pictures (i.e. pixel files) on the secondary optical disk. Then, the header information of a large set of files could be browsed to find the desired image before the pixels are brought from secondary to primary storage. This scheme would trade a few seconds of access time on the first access for the savings of having optical rather than magnetic disk drives.

In short, there are times when both separated and combined image files are useful. Therefore, we decided to support both types of storage.

In recent years, there have been increasingly more situations in which one might want to store multiple images for simultaneous processing. For example, within the medical imaging community there is currently a great interest in scale-space techniques for segmentation of images.[9] For this reason, we sought to support multi-image files. As a convenience, we refer to these as "stacks" and call each separate picture, regardless of its number of dimensions, a "level." We chose to permit the levels to be completely independent of each other. Thus they may have entirely different dimensions, pixel types and associated information.

In keeping with our decision to allow storage of images in separated and in combined files, we decided to permit stacks to be stored this way. However, we also believe a more natural method of storing each level in a separate file should be supported. For this purpose we support the concept of a directory of images. For a typical application, a directory would be created to hold the image. In it, a single header file and multiple

† In this paper we use the term "pixel" regardless of image dimension.

pixel files would be placed. This directory of files then would be processed as if it were a single image stack. The single header file would contain the descriptive information for each picture. We also allow there to be multiple header files in the same stack.

One of the nice features of the UNIX system is the idea of pipes that connect separate processes called filters.[10] A filter takes data from its standard input, does some computation on the data stream and writes the resulting output to its standard output. The vertical bar, which indicates a pipe, allows such programs to be connected to form longer commands. This idea has generally not been applied to images, because of their size. However, tests at UNC have shown it is feasible on today's workstations for all but the largest of images. This is particularly appropriate when a user wants to test a number of algorithms on the same image and look at the results on a display screen. For example, the commands

```
% ahe chest | display &
% butterworth chest | display &
% median chest | display &
```

might run three well known enhancement techniques on the same image (chest), all being run in the background (the ampersands), so that they could be displayed simultaneously side-by-side. No intermediate files, which the user would later have to delete, would be produced.

This concept of piping was extended to other objects. Thus, in addition to piping images, we decided to support memory sharing between processes with the connection supported via pipe. To avoid piping image data, we also permitted "references" to images to be piped.

The idea of a reference to an image is an important part of our design. This is of particular importance in medical fields, where a physician, for example a radiologist, often may want to compare similar images from several patient studies to improve his understanding of how a particular anatomical feature or abnormality should appear. It would be too tedious to have to merge the images, prior to display, and then issue commands to display them. With reference files, which contain lists of image file names, one can define new images, especially stacks, without actually copying files. Therefore, the I/O software must to be able to interpret references, so that the actual work can be done at the time of final execution.

Because some of our clinical colleagues were strong supporters of the ACR-NEMA message format, they urged us to base our format on it. While this idea had merit, the ACR-NEMA standard has some serious limitations. Among them are that it supports 1D and 2D images only, and it has a limited number of pixel types. However, the mechanism for defining new group-element pairs is attractive. Therefore, we decided to use the essential form of ACR-NEMA, that is (group#, element#, value) in our header and to implement as many of the ACR-NEMA group-element pairs as possible. However, we also decided to permit an unlimited number of image dimensions and to define pixels based on all primitive types plus common aggregate types (e.g., binary, complex and color). Moreover, from past experience, we knew this would not always be adequate, so we defined a mechanism whereby a programmer can define new pixel types as aggregates of the primitive types.

We did not adopt the ACR-NEMA scheme to store image pixels and overlay data (coincident text or graphics used for annotation of the image). The fact that ACR-NEMA specified a message format, implying serial access, makes it inappropriate for disk storage where one wants direct access for the fastest possible I/O. Moreover, since we wanted to support image compression, we were forced to consider an alternative scheme for storing pixels. Since overlays can be quite large, we decided to store them as we do pixels.

Because it uses the basic ACR-NEMA scheme for image header data, our format supports an essentially unlimited number of descriptive items. To make this capability convenient to use, we developed a data dictionary to translate a list of key names into group-element pairs. Thus, programmers using our software would not need to know anything about the ACR-NEMA standard; they only need to select the appropriate key names for the data items they want to store or retrieve.

We included other important features in our design. For example, to make it easy for a user to customize the system to meet particular requirements, we specified files and environment settings that provide run-time control of the system. Similarly, we defined a sophisticated mechanism for controlling the output of messages generated by the system to the user, to deliver as little or as much information as desired.

Finally, we chose an object-oriented approach for the software construction because of the size of the project. We agree with others that at present the best, practical, object-oriented language for systems development in UNIX is C++.[11] The object inheritance capabilities of C++ added flexibility to the implementation. This allows us to extend the capabilities of the software to meet unanticipated future needs by deriving new objects, such as new image types.

5. BASIC SYSTEM COMPONENTS

The software package implied by this design must consist of two primary components: a set of supporting libraries and a set of programs based on the libraries. A wide variety of programs for image processing, computer vision, neural network simulation and other purposes can be built on the set of libraries we are developing. The remainder of this paper describes the library architecture. The set of programs (/usr/image) we are using and will continue to develop is not addressed. The only program that must always accompany the libraries is the dictionary compiler, explained below. The appendix presents the formal grammar for the file format.

The present design calls for eight libraries, although since the implementation is not complete at this writing, that may change. In particular, the code for handling shared libraries, which has not yet been designed, may be put into an additional library. The libraries defined at present are as follows:

libbase.a	A base library containing functions common to all programs. This is always used, since the other libraries depend on it.
libhdr.a	A library for encoding and decoding arbitrary data by use of dictionaries to form the "header" portion of images.
libio.a	The I/O library, which transfers headers, pictures and overlays between memory and disk. This also handles issues of pixel encoding.
libcompr.a	A collection of compression functions available to the I/O library. Use of these may easily be specified by the user at run time.
libbuf.a	A library for access to and manipulation of pixels in memory.
libtype.a	A collection of conversion functions used by the buffer library to convert images from one type to another as required by the application.
libimg.a	A library that couples the I/O library and the buffer library, so that images can be dealt with as objects.
libcompat.a	A compatibility library to allow the existing /usr/image programs to be used with the new file format. This implements a conversion technique and will eventually be discarded.

The reason for having so many libraries is two-fold. It provides more flexibility and speed to application programmers because only libraries that are needed are linked with application objects to form programs. Entire libraries, especially the compression and type conversion libraries, can easily be replaced for special applications. The other reason is that it makes the software design cleaner and more maintainable. The hierarchy of this architecture encouraged us to decide carefully into which library the various components

had to go.

The following paragraphs discuss selected components of the package and indicate how they all fit together.

5.1. The dictionary compiler

The dictionary compiler (dcomp) implements a formal language called Dictionary Compiler Language (DCL). Dcomp translates header key specifications written in DCL to produce a binary file called a dictionary and a C++ include file (dict.h), which defines the set of keys used to index the dictionary. The values in the dictionary are the formats and default values corresponding to the header keys. The use of format and default value specifications originated with the ACR-NEMA standard, although our implementation is more flexible (Appendix, Section 3).

A set of functions is also provided for searching the dictionary. For example, the following C++ statements would store and then retrieve the number of dimensions for level 3 of the image associated with class instance Image2:

```
int  NumDim;      // Declaration of variable to receive result.
return_val = Image2.GetKey(szIMAGE_DIMENSIONS, NumDim, 3);
return_val = Image2.PutKey(szIMAGE_DIMENSIONS, NumDim, 3);
```

The character string szIMAGE_DIMENSIONS is an example of a header key that corresponds to a group-element pair. This key is defined automatically in the dict.h file produced by the dictionary compiler.

As many as three dictionaries may be simultaneously used at a particular site. They are the site dictionary, which provides the majority of keys, the group dictionary, which defines project-specific keys, and the user's own private dictionary defining keys only he is using. The dictionaries are merged at the time of execution.

While it will not be usual to do so, it is possible to store the dictionaries at the beginning of an image file. This will be of benefit when one needs to send an image to a remote site. Thus, in the same sense as with the HDF format[6], our format is self-describing. Just as in that case, software based on the special keys would have to be used to do anything meaningful with these keys.

5.2. I/O processing

The figure shows how several of the libraries fit together. Some of the boxes within each library are functional units, not necessarily C++ classes or modules. I/O can be thought of as proceeding from a request at the top of the tree down through the various layers to the "System" box at the bottom right, which represents the actual interface to UNIX.

Consider, for example, what happens when one reads a file. An instance of the image file class will be constructed for pixels of the desired type. The header data will then be read from disk and interpreted by the header class (calling the header library) using the appropriate dictionaries. This provides all the information necessary for interpreting the pixel data. If the pixels are the same type as required for the application, they are read directly into the pixel buffer. If they are another type, the appropriate functions from the type conversion library are used. XDR decoding and byte swapping are done in the I/O library. If the pixels on disk are compressed, the I/O library calls the appropriate function in the compression library to uncompress them. We make it possible for a level to be broken into "chunks" for storage on disk, so it is often unnecessary to uncompressed an entire file to read part of it.

To produce an output file, a similar approach is used. However, in this case, it is possible to provide the user greater control over the file format. For example, here is a possible sequence of commands given by

a user to produce two files containing the same pixel data.

Fig. The arrangement of five of the libraries used to support an application program is shown. The header library (libhdr.a) and base library (libbase.a) are not shown, although required. The header library is called from the header class in libio.a, and the base library supports all libraries.

```
% setenv XDR_MODE RAW
% setenv COMPRESSION_ALGORITHM NONE
% makeimg file_1

% setenv XDR_MODE FULL
% setenv COMPRESSION_ALGORITHM RLE
% makeimg file_2
```

In the first case, the user has specified in the environment that XDR encoding will not be used and the pixels will not be compressed. In the second case, the pixels will be XDR encoded and run-length encoded (RLE). Typically, such environmental variables will be stored in a file called .imagerc in the user's home directory. But for quick changes of the settings, the environment variables override the contents of this file. The point here is that only a single executable program (makeimg) is needed, and there is nothing special about it that makes it work with XDR, compression or any of the other features of the libraries.

The hierarchy of the image library (libimg.a) extends the basic capability of the lower-level libraries to references, directories and pipes. For example, a "file set" can be viewed as consisting of either a single image (whether in one or two files), a directory of several files (stack) or a piped image. Typically, application programs will only interact with the libraries at the "image stream" level, so that the full capability of the system will be available to users.

6. PROGRESS TO DATE

The software is written with version 2.1 of AT&T's C++[12]. The developed code is running on workstations manufactured by Digital Equipment Corporation, the Hewlett-Packard Company and Sun Microsystems, Inc. Eventually we plan to make the software work on a wider selection of compilers and machines.

The programs presently using the new libraries are simply the approximately 100 programs that comprise /usr/image. The compatibility library allows us to use the old programs with the new format by simply recompiling them. A limited subset of the features of the new software are then available to the old programs. By use of the compatibility library, a smooth migration from the old format to the new one is possible. The compatibility library can read and write in either format, although it usually writes in the new format only. This assures that existing files will gradually be converted to the new format. As the new libraries stabilize and programmers become more familiar with them, we expect to replace many of the older programs with new versions.

Most of the code for the eight libraries described above has been written. However, a few major components, especially pipes and directories have not been implemented yet. The reference capability is only partially completed. In particular, a supporting tool to be called "Database Management Program" (DMP) will be written to provide a convenient interactive means of editing references. This will be described in more detail at a later date in a technical report by members of the University of Utrecht 3D Computer Vision Group.

At present, the source code consists of about $22,800$ lines in .c files, with $7,000$ lines of definitions in .h files. In comparison, the older libim.a library contained $1,700$ lines of C code. Because the new package is large, one might reasonably ask how it performs.

A simple test was run to indicate the performance differences in the old and new libraries. A 3D image with dimensions $256 \times 256 \times 109$ and 16-bit integer pixels was used. The pixels were stored in native machine byte order without using XDR. Only a few keys were used in the header. The test consisted of reading the image and computing its 20-bucket histogram. The calculation of the histogram was done with the same program, so the primary effect was to measure the relative input transfer rates. The test was made on a Sun Microsystems SPARCstation 1+ (15.8 mips) workstation. The average times from ten runs using the

old library and format were 13.41 sec. for application code and 7.85 sec. for operating system code. The averages of ten runs from the same program compiled for use with the new format and using the new file format were 19.48 sec. for application code and 12.99 sec. for operating system code. This represents a 50% increase in time, when using the new software.

This might seem discouraging were it not for the fact that no execution-time profiling has yet been done on the code. It is a "first draft" result; we are confident that after we have had a chance to determine in detail where the execution time is being spent, we will be able to reduce the time to almost the same as in the considerably more compact old library.

Moreover, as workstations today are many times faster than the original VAX† 11/70 computer for which the old libim.a library was written, we are willing to trade a small loss of computational speed for the tremendous increase in function described here.

7. ACKNOWLEDGEMENTS

This work was partially supported by NIH grant #P01 CA47982 and SPIN grant (VS-3DM) 50249/89–01 from the Dutch Ministry of Economic Affairs. The authors wish to thank James Coggins, Brad Hemminger, Rick Lonon and Stephen Pizer for their valuable discussions and encouragement. We are also grateful to Mark Kupper who wrote part of the type conversion library.

APPENDIX

Three grammars that specify the construction of the file formats are given below using extended BNF. The full grammar for the system will be published in a UNC technical report at a later date.

1. Combined file grammar

An `<image_data>` file, which contains both the header and picture components, is referred to as a combined file. Its organization is described by the following grammar.

```
<image_data>    ::= <prelude> <transport> <header> <picture> [<overflow>]
<prelude>       ::= <handshake> <version> hdr_addr hdr_len ofl_addr ofl_len
<transport>     ::= [[[<site_dict>] <group_dict>] <user_dict>]
<header>        ::= [<hdr_body> [<reserve>]]
<picture>       ::= [[<gap>] <chunk>]* [<gap>]
<handshake>     ::= <stream_code> NUL NUL <byte3>
<stream_code>   ::= NUL
<version>       ::= v1 v2 v3 v4
<hdr_body> <overflow>  ::= <hdr_string>
```

The `<stream_code>` indicates whether a stream is an image, reference or shared memory key. For an image, it is the ASCII `NUL` character. The single characters `v1` – `v4` are treated as integers with range [0, 255] to specify a 4-part version number for the file format.

The non-terminals `<user_dict>`, `<group_dict>` and `<site_dict>` represent the three dictionaries used to make the file. The `hdr_addr`, `hdr_len`, `ofl_addr` and `ofl_len` are 4-byte integers stored serially. If present, they specify the address and length in bytes of the header body and overflow regions. The existence or non-existence of each of the three dictionaries, of the `<hdr_body>` and `<overflow>` strings and of the corresponding addresses and lengths is encoded into `<byte3>`. In practice the header body is always present. Notice that the `<hdr_body>` and `<overflow>` strings are not defined on their own. Rather, they are concatenated to form the header string. This is then used in grammar 3, below.

† VAX is a registered trademark of Digital Equipment Corporation.

A <chunk> is a string of bytes that contains encoded pixels or an overlay. A level may contain multiple chunks. A gap is an unclaimed sequence of bytes, typically all NUL. Gaps are usually omitted, but can be used for adding additional chunks or for application-specific purposes. The <reserve> is an unclaimed sequence of bytes, typically all NUL, used for expansion of the header.

2. Separated files grammar

The alternate way of storing images is in separated files. In this case, the <image_data> is broken into two parts, the header_data and the picture_data. Their contents are described by the two grammars given below. In this case the <overflow> string vanishes. Because the elements used in this grammar are identical to those in the combined file grammar above, their definitions are not repeated here.

```
header_data    ::= <prelude> <transport> <header>
<prelude>      ::= <handshake> <version> hdr_addr hdr_len
<transport>    ::= [[[<site_dict>] <group_dict>] <user_dict>]
<header>       ::= [<hdr_body> [<reserve>]]
<handshake>    ::= <stream_code> NUL NUL <byte3>
<stream_code>  ::= NUL
<version>      ::= v1 v2 v3 v4
<hdr_body>     ::= <hdr_string>

picture_data   ::= <picture>
<picture>      ::= [[<gap>] <chunk>]* [<gap>]
```

In this case the <hdr_body> and the <hdr_string> are the same thing. The next grammar explains how the <hdr_string> is constructed.

3. Header string grammar

The <hdr_string> is an XDR encoded stream of data having an organization similar to the ACR-NEMA message format. All terminals are encoded by XDR before being stored in the string.

```
<hdr_string>   ::= number_of_items <items>
<items>        ::= [<item>]*
<item>         ::= group element <groupdata>
<groupdata>    ::= <sep_data> | <def_data> | <glob_data>
<sep_data>     ::= num_pairs [index data]*
<def_data>     ::= <default> num_pairs [index data]*
<glob_data>    ::= data
<default>      ::= one | zero data
```

A header string consists of number_of_items items. Each item contains an ACR-NEMA group-element pair, the group and element terminals, and a data item. The three kinds of data items are separate, default and global. The num_pairs terminal is a decimal number indicating the number of pairs to follow. The index is a level number in the stack, and data is the information to be stored under the given group-element combination for the designated level. A global key has only a single data item that applies to all levels, and a default key has a default data value to be used for all levels not having a specific value. A separate key is expected to have a data value for every level. The type applicable to <groupdata> is stored in the dictionary and is implied by the group and element numbers. Terminals one and zero are the decimal values 1 and 0, respectively. They indicate the presence/absence of the default value.

8. REFERENCES

1. John Zimmerman, George Entenman, Mike Fitzpatrick and Jane Whang, "V Shell Reference Manual," Department of Computer Science, University of North Carolina at Chapel Hill, May 1981.

2. Sun Microsystems, Inc., "XDR: External Data Representation Standard," RFC 1014, DDN Network Information Center, June 1987.

3. National Electrical Manufacturers Association, Digital Imaging and Communications, ACR-NEMA Standards Publication No. 300–1985, 2101 L Street, N.W., Washington, D.C.

4. John Rasure and Carla Williams, "An Integrated Visual Language and Software Development Environment," *J. Visual Languages and Computing*, vol. 2, pp. 217–246, 1991.

5. R. A. Robb and D. P. Hanson, "ANALYZE: A Software System for Biomedical Image Analysis," First Conference on Visualization in Biomedical Computing, pp. 507–518, IEEE Computer Society Press, Los Alamitos, CA, 1990.

6. NCSA Software Development, NCSA HDF Specifications, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana IL, March 1989.

7. James M. Coggins, "Integrated Class Structures for Image Pattern Recognition and Computer Graphics," Proc. USENIX C++ Workshop 1987, pp. 240–245, USENIX Association, Berkeley CA, 1987.

8. D. C. Wells, E. W. Greisen, and R. H. Harten, "FITS: A Flexible Image Transport System," *Astron. Astrophys. Suppl. Ser.*, vol. 44, no. 3, pp. 363–370, June 1981.

9. Harry Wechsler, Computational Vision, Academic Press, Boston, 1990.

10. Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, July 1974.

11. Kim Polese and Ted Goldstein, "OOP Language & Methodologies," *SunWorld*, vol. 4, no. 5, pp. 44–61, May 1991

12. Bjarne Stroustrup, The C++ Programming Language, 2nd. ed., Addison-Wesley, Reading MA, 1991.