INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

VIDEO++: AN OBJECT-ORIENTED APPROACH TO VIDEO ALGEBRA

by

Ghassan Hammouri, B.Eng.

A thesis submitted to the

School of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical Engineering and Computer Engineering

School of Information Technology and Engineering (Electrical & Computer Engineering)

Faculty of Engineering
University of Ottawa

July, 1997

Abstract

Video has become an important medium in the field of multimedia communications. An increasing amount of video processing software is being developed to manipulate video (and audio) in an efficient and easy-to-use manner. Great advances have been made in the fields of video compression, video indexing, and content-based retrieval but, in spite of its growing importance, video has still not been endorsed as a *data type* in the software development community. The main reason is that video, unlike other data types such as integers and strings, is complex and difficult to manipulate. Its size prohibits it from being loaded into the random access memory of most computers, its spatial structure is complex, and the temporal order of its frames must be preserved at all times.

This thesis proposes a new concept, video algebra, which allows developers to manipulate video as basic data types. This thesis also presents a new method for structuring video which is used by video algebra. A prototype has been developed in the form of an object-oriented framework called **Video++**.

Video++ introduces two new concepts; the video markup language (VML) and the video description header. VML is a markup language that describes video segments and their contents. The video description headers serve three purposes. First, they complement the visual information in video segments by providing text annotations written in VML. Second, they provide a mechanism for reusing existing video segments. Third, they provide the constructs that are required when algebraic operations are applied to video segments.

The Video++ prototype is written in C++ and implements most of the algebraic and logical operations that can be found for simple data types. Test results have shown the Video++ system to be fast and efficient. Complex algebraic operations are executed almost instantaneously and the reuse of video segments saves on disk space. Content-based search, using the textual information in the video description headers, is fast and straightforward.

List of Figures

Figure 1: The format of a raw video segment 8
Figure 2: Multimedia Storage Requirements 11
Figure 3: Bidirectional Prediction in MPEG 12
Figure 4: Objects interacting via messages 13
Figure 5: The effect of object-orientation on development effort 14
Figure 6: The learning curve for mastering C++ 16
Figure 7: The MIT Algebraic Video System 18
Figure 8: Two sample video segments 25
Figure 9: Addition of the two video operands 26
Figure 10: Subtraction of video 2 from video 1 27
Figure 11: Subtraction of video 1 from video 2 27
Figure 12: The union of video1 and video 2 27
Figure 13: The intersection of video 1 and video 2 28
Figure 14: The intersection of video 2 and video 1 28
Figure 15: The unisection of video 1 and video 2 29
Figure 16: Two video segments which are not equal 30
Figure 17: The subset of a video segment 30
Figure 18: The superset of a video segment 31
Figure 19: The subscript operation 32
Figure 20: The range operation 33
Figure 21: A video segment with inter-video frame sequences 35
Figure 22: A stratagraph 42
Figure 23: The duality of video 43
Figure 24: A tree showing terminal and nonterminal vertices 51
Figure 25: Tree Addition 52
Figure 26: Tree Subtraction 53
Figure 27: Tree Union 53
Figure 28: Tree Intersection 53
Figure 29: Tree Unisection 54
Figure 30: Tree Equality 54
Figure 31: Tree Subset 55
Figure 32: Tree Superset 55
Figure 33: A video dependency tree 56
Figure 34: Video Algebra using VML 58
Figure 35: The structure of a news program 64
Figure 36: The waterfall method and the object-oriented, iterative method 67
Figure 37: The video class at the analysis stage 71
Figure 38: The vistream and vostream classes 72
Figure 39: The compressor and decompressor classes 73
Figure 40: The compression inheritance tree 74
Figure 41: The video class at the design stage 77
Figure 42: The VideoDesc class 79
Figure 43: Tree representation using pointers 80
Figure 44: The Node class 81

Figure 45: The VideoTree class 83
Figure 46: The conceptual video tree model 84
Figure 47: The video++ module diagram 96
Figure 48: An alternative video class diagram 100

List of Tables

Table 1: Various bit rates for multimedia types 6
Table 2: The MIT Algebraic Video Operations 20
Table 3: Video algebra operations 24
Table 4: VML Symbols 38
Table 5: Frame expressions 40

Table of Contents

1. INTRODUCTION 1

1.1 THE DEFINITION OF THE PROBLEM 1
1.2 THESIS ORGANIZATION 2
1.3 MAIN CONTRIBUTIONS 2
1.4 TERMINOLOGY 3

2. REVIEW OF BASIC CONCEPTS 4

2.1 MULTIMEDIA 4
2.2 DIGITAL VIDEO CONCEPTS 5
2.2.1 Size 5
2.2.2 Content 6
2.2.3 Dimension 7
2.2.4 Format 8
2.3 IMAGE RETRIEVAL 9
2.3.1 Text-Based Retrieval 9
2.3.2 Content-Based Retrieval 9
2.4 IMAGE INDEXING 10
2.5 VIDEO INDEXING 10
2.6 VIDEO COMPRESSION 11
2.7 THE OBJECT-ORIENTED PARADIGM 13
2.8 THE C++ PROGRAMMING LANGUAGE 15
2.9 SUMMARY 16

3. THE MIT ALGEBRAIC VIDEO SYSTEM 17

3.1 ARCHITECTURE 17
3.2 CONTENT-BASED ACCESS 19
3.3 VIDEO ALGEBRAIC OPERATIONS 19
3.4 SUMMARY 22

4. DESCRIPTION OF VIDEO++ 23

4.1 VIDEO ALGEBRA 23
4.1.1 Addition 26
4.1.2 Subtraction 26
4.1.3 Union 27
4.1.4 Intersection 28
4.1.5 Unisection 29
4.1.6 Equality 29
4.1.7 Inequality 29
4.1.8 Subset 30
4.1.9 Superset 30
4.1.10 Compression 31
4.1.11 Decompression 32
4.1.12 Subscript 32
4.1.13 Range 33

4.1.14 Search *33*
4.2 VIDEO++ 34
4.3 THE VIDEO MARKUP LANGUAGE (VML) 36
4.3.1 HyTime *37*
4.3.2 VML syntax *38*
4.3.3 Alternative description languages *41*
4.4 THE VIDEO DESCRIPTION HEADER (VDH) 42
4.4.2 VDH Storage *47*
4.4.3 Video Distribution *48*
4.5 FURTHER VDH APPLICATIONS 48
4.5.1 The CCIR Universal Header/Descriptor *49*
4.5.2 Audio Support *49*
4.5.3 Video Indexing *50*
4.5.4 Video-On-Demand Databases *50*
4.6 TREES AND TREE ALGEBRA 50
4.6.1 The Tree Data Structure *50*
4.6.2 Tree algebra *52*
4.6.3 The Video Tree *55*
4.7 VIDEO ALGEBRA USING VML 56
4.7.1 Addition *58*
4.7.2 Subtraction *59*
4.7.3 Union *60*
4.7.4 Intersection *60*
4.7.5 Equality *60*
4.7.6 Subset *61*
4.7.7 Superset *61*
4.7.8 Subscript *62*
4.7.9 Range *62*
4.7.10 Compression *63*
4.7.11 Decompression *63*
4.7.12 Search *63*
4.8 CASE STUDY 64
4.9 SUMMARY 65
5. THE DEVELOPMENT OF VIDEO++ 66
5.1 PROBLEM DOMAIN AND REQUIREMENTS ANALYSIS 67
5.1.1 Video Algebra *68*
5.1.2 Frames *68*
5.1.3 Input/Output *68*
5.1.4 Compression/Decompression *68*
5.2 ANALYSIS 69
5.2.1 Video Algebra *69*
5.2.2 Frames *71*
5.2.3 Input/Output *71*
5.2.4 Compression/Decompression *73*
5.3 DESIGN 74
5.3.1 Video Algebra *75*

5.3.2 *The Video Tree* 79
5.3.3 *Object Creation and Access* 84
5.3.4 *Frames* 87
5.3.5 *Input/Output* 87
5.3.6 *Compression/Decompression* 93
5.3.7 *Error Handling* 93
5.3.8 *The Module Diagram* 95
5.4 IMPLEMENTATION 96
5.4.1 *Platforms* 97
5.4.2 *Class libraries* 97
5.4.3 *Programming* 98
5.4.4 *The Berkeley Compressor/Decompressor* 98
5.5 TEST RESULTS 100
5.6 SUMMARY 104

6. COMPARISON WITH THE MIT AV SYSTEM 105

6.1 SIMILARITIES 105
6.1.1 *Hierarchical Video Structure* 105
6.1.2 *Video Content* 105
6.1.3 *Video Algebraic Expressions* 106
6.1.4 *Video Format* 106
6.1.5 *Development and Implementation* 107
6.2 DIFFERENCES 107
6.2.1 *Video Storage* 107
6.2.2 *Algebraic Operations* 107
6.2.3 *Video Output* 108
6.2.4 *Video Content Descriptions* 108
6.2.5 *Interface and Usage* 108
6.3 SUMMARY 109

7. CONCLUSION & FUTURE RESEARCH 110

7.1 FUTURE RESEARCH 110
7.1.1 *Text retrieval* 110
7.1.2 *Front-end GUI* 111
7.1.3 *Real-Time Video* 111
7.1.4 *Standard Video++ format* 111
7.1.5 *MPEG compressor/decompressor* 112
7.1.6 *Remote Access* 112
7.1.7 *Pipelining* 112
7.1.8 *Device Drivers* 113

8. APPENDICES 114

9. REFERENCES 116

1. INTRODUCTION

Video has become an important medium in multimedia communications. Standard text-based e-mail is being replaced by other standards which support multimedia formats. Teleconferencing and video-on-demand cable systems also depend on video and audio. However, video will only become an effective part of everyday computing environment when we can use it with the same ease as we currently use text. Word processors and desktop publishing systems are sophisticated tools for manipulating and editing text. Similar composition using video remains a challenging goal [19].

There are a number of characteristics of video which prevent it from reaching its potential for expressive power, creativity, and communications. Among them are a lack of manipulation tools and technologies for the average user [30]. Most video processing software packages are fairly complex and cannot be easily integrated with other development packages. Furthermore, the atomic elements of video, the frame and the sequence, are of such coarse grain that, without the proper tools, complete control is impossible to anyone but the original creative production, whereas the symbolic elements of text - words - can be edited by anyone [21]. This thesis shows that, to handle video effectively, developers must have at their disposal the same algebraic and logical operators that are currently available for simpler data types. With the availability of these operators, developers can use familiar algorithms and techniques to work with video data. This thesis proposes a novel way of structuring video that lends itself to algebraic and logical expressions. It also defines a complete set of algebraic and logical operators that can be applied to video. This set constitutes what we refer to as video algebra. Video algebra defines unary and binary operators that take as input one or two video segments and return a result. Algebraic expressions may be combined to form more complex expressions.

1.1 The Definition of the Problem

Most video processing software exists in the form of either libraries or stand-alone packages. Libraries usually export a set of routines that the users may use in their application programs. These libraries and stand-alone packages may allow direct manipulation of frames,

compression ratios, and color bits, but very few of them take video *content* into consideration. If we wish to compose with video as we currently compose with words, we must focus our attention on content. Video composition should not entail thinking about pixels any more than text composition requires thinking about ASCII character codes [19]. However, even state-of the-art software for manipulating video cannot extract and interpret all the content information available in a video segment. This kind of software would, instead, work with time codes, individual frames, and clips of video and sound [19]. Thus video indexing and retrieval deals with abstractions of video content, and the most familiar abstraction is that of text [19].

This thesis proposes a solution to two problems pertaining to video, as discussed above. First, it solves the problem of extracting and interpreting video content by placing textual content description inside video description headers. Second, it introduces a new way of structuring video. This new structure allows video to be used in algebraic expressions. To implement this solution, a C++ class library called Video++ has been developed using an object-oriented paradigm.

1.2 Thesis Organization

The thesis is organized as follows. The basic concepts of the object-oriented paradigm, C++, and video algebra are presented in Chapter 2. The MIT Algebraic Video system is described in Chapter 3. The theoretical aspects of Video++ are detailed in Chapter 4. Chapter 5 outlines the implementation of Video++. Comparisons with the MIT AV system are then discussed in Chapter 6. The conclusions and suggestions for further research are presented in Chapter 7.

1.3 Main Contributions

The main contributions of this thesis are:

- Definition of a complete set of algebraic operators for video.
- Proposal of a new method for implementing a video algebra by defining the concept of video description headers and video dependency trees.

- Implementation of the proposed video algebra using C++.
- Demonstration of the complete development cycle for writing the Video++ class library using an object-oriented approach.

1.4 Terminology

The following terminology is used throughout this thesis:

- The terms **video**, **stream**, **segment**, and **movie** are used interchangeably.
- A video segment is considered **active** if it is currently being played back in a window. Otherwise, the video segment is considered **inactive**.
- The term **content analysis** is used to mean the process involved in 'understanding' the content of a video stream. Traditionally, this process usually involves algorithms taken from the fields of image processing, artificial intelligence, and pattern recognition.
- The terms **simple data types**, **regular data types**, and **intrinsic data types** are used interchangeably. Intrinsic data types are built into most programming languages. For languages such as C and Pascal these intrinsic data types are usually integers, real numbers, characters, and pointers. Smalltalk, however, does not have intrinsic data types; all data types are *classes* which describe objects.
- The term object is sometimes used to mean "instantiated class". For example, a **video object** is instantiated from the **video class**.
- The terms **video tree**, **dependency tree**, and **video dependency tree** are used interchangeably.

2. REVIEW OF BASIC CONCEPTS

This chapter will cover the basic concepts pertaining to multimedia, video, object-oriented software design, and the C++ language.

2.1 Multimedia

Multimedia is an interdisciplinary, application-oriented technology that capitalizes on the multisensory nature of humans and the ability of computers to store, manipulate, and convey nonnumerical information such as video, graphics, and audio in addition to numerical and textual information [33]. It is not a single technology but rather a class of technologies and applications that span two or more information types such as voice, data, video, and graphics. The Telecommunication Standardization (TS) sector of the International Telecommunications Union (ITU) defines multimedia as an application involving at least *two* distinct information types [33]. The two information types that are most significant to this thesis are text and digital video.

An all-digital approach to multimedia communication merges the applications and technologies in the computing, office automation, and consumer electronics areas with those in the telecommunications area. Multimedia systems require synchronization of some of the information types in communication, processing, and presentation, and in the longer term will be characterized by the completely digital integration of all information types [4].

Hypermedia, as an extension of hypertext incorporating information types other than text, denotes an organizational concept of nonlinear network-like structuring of multimedia documents. It provides computer-supported nonsequential authoring and reading, i.e. interactive branching and dynamic display using information objects and multichoice links in between. Hypermedia documents enable the reader to navigate along defined multiple paths through related information material, annotations, figures, film scenes, bibliographical data, etc. within an information network of links and nodes [4].

2.2 Digital Video Concepts

An important goal of digital video is to integrate it into the computer [8]. This paradigm provides random-access editing, inter-activity, image processing, seamless integration with computer graphics, integration into various types of computer documents, storage on hard drives and CDs, and transmission across networks and phone lines [8]. Video can be seamlessly integrated into the computer by making it available as a standard data type in some well known programming language. As a data type it could then be easily manipulated by developers. To implement video as a data type, however, it is important to understand the complexities of video; video has several properties which distinguish it from the other data types such as integers and strings.

2.2.1 Size

There are two problems with digital video; it has a large size and the size is not fixed (i.e. it varies from one video segment to another). In contrast, integer sizes are small and fixed (usually 32 bits on most UNIX systems).

For example, let us assume that we have a two-hour video segment played at full-screen mode. The video segment would play for 2×3600 seconds at 30 frames a second. At full screen mode, a frame would consist of 1280×1024 pixels, the maximum resolution on many standard workstation monitors. One single pixel is represented by three bytes in memory, assuming a 24-bit color scheme. Thus the entire video segment would consume (2×3600×30)×(1280×1024)×24 = 6.8 trillion bits (or 800 gigabytes). A real-life example is Disney's Toy Story (released 1995) which is the first full-length animated movie created entirely using computers. Running 77 minutes, it consisted of 110,064 frames and consumed over 500 gigabytes of disk space.

Transmission of video also requires more network bandwidth than other data types. In some cases, networks must provide a variety of constant, variable, and burst-oriented bit rates of up to 100 Mb/s and more per connection.

Information Type	Bit rate	Quality and remarks
Data	Wide range of bit rates	Continuous, burst-and packet-oriented data
Text	Some Kb/s	Higher bit rates for downloading of larger volumes
Graphics	Relatively low bit rates	Depending on transfer time required
	100 Mb/s or more	Exchange of complex 3D computer models
Image	64 kb/s	Group-4 telefax
	Various	Corresponding to JPEG standard
	Up to 30 Mb/s	High-quality professional images
Video	64-128 Kb/s	Video telephony (reduced quality; CCITT H.261)
	384 Kb/s - 2 Mb/s	Video conferencing (reduced quality; CCITT H.261)
	1.5 Mb/s	Today's quality of video cassette recorders (MPEG 1)
	5-10 Mb/s	Standard TV quality (MPEG 2)
	34/45 Mb/s	TV contribution
	50 Mb/s or less	HDTV distribution
	100 Mb/s or more	Studio-to-studio HDTV in top quality; time-compressed video downloading
Audio	n×64 Kb/s	3.1 kHz, 7 kHz, of hi-fi baseband signals (CCITT)

Table 1: Various bit rates for multimedia types

The typical bit rates to be expected for the various information types within multimedia communication are compiled in the table above. Particular importance is attached to existing and emerging standards for image and video coding for a number of purposes. These range from interactive video communication to video storage, retrieval, and processing to video distribution [4].

2.2.2 *Content*

Video is extremely rich in content. It is impossible to associate a single value to a video segment and, furthermore, video content is usually subject to interpretation by the observer. Querying a video stream by content remains a daunting task. With the current technology, even sophisticated pattern recognition and feature selection algorithms do not always match common human perception.

Some of the problems posed by automatic content analysis for video are:

- None of the existing algorithms are powerful enough to correctly extract all the features in a video frame.
- None of the algorithms address the complex interdependencies between frames in a video segment.

- Even if all the features were correctly extracted, video content would still depend on the individual judgment of the classifier. Computers could not be expected to comprehend the historical and cultural implications that pervade so many of our images and videos.

Even simple data types such as integers are subject to interpretation. Little-endian architectures and big-endian architectures associate different values to the same contents in memory. In the little-endian architecture bytes at lower addresses within a given 16- or 32-bit word have lower significance (the word is stored 'little-end-first'). Intel microprocessors and most communications and networking hardware are little-endian. In a big-endian architecture the most significant byte within a given multi-byte numeric representation has the lowest address (the word is stored 'big-end-first'). The Motorola microprocessor families, and most of the various RISC designs, are big-endian [18].

With video, however, the information is overwhelmingly more complex and the proper interpretation of the contents becomes even more difficult. As will be shown later, Video++ relies on video description headers to extract and interpret video content.

Video is sometimes accompanied by at least one other medium that helps in the interpretation of its contents. The two most common mediums are sound and text. Words (whether in the form of sound or text) can provide human observers with valuable information that can put video in its proper context. The Video++ system uses textual annotations for the description of the video contents.

2.2.3 Dimension

Video has two dimensions. It occupies a physical portion of the screen thus giving it a *spatial* dimension. It also runs for a finite period of time thus giving it a *temporal* dimension. Still images have only a spatial dimension while audio has only a temporal dimension. Intrinsic data types such as integers are not considered to have any dimensions. All video input/output operations must take both dimensions into consideration.

2.2.4 *Format*

For all practical purposes, digital video is considered to be a sequence of frames, each frame consisting of a rectangular array of pixels.

A shot is a continuous video signal. The camera remains ON during the duration of the shot and can go through one of seven motions; tracking, dollying, booming, zooming, panning, tilting, or remain fixed. Shots are separated by cuts [47]. A cut is not necessarily an abrupt camera break; there are more sophisticated cuts that involve gradual changes between consecutive frames such as dissolve, wipe, fade-in, and fade-out [7].

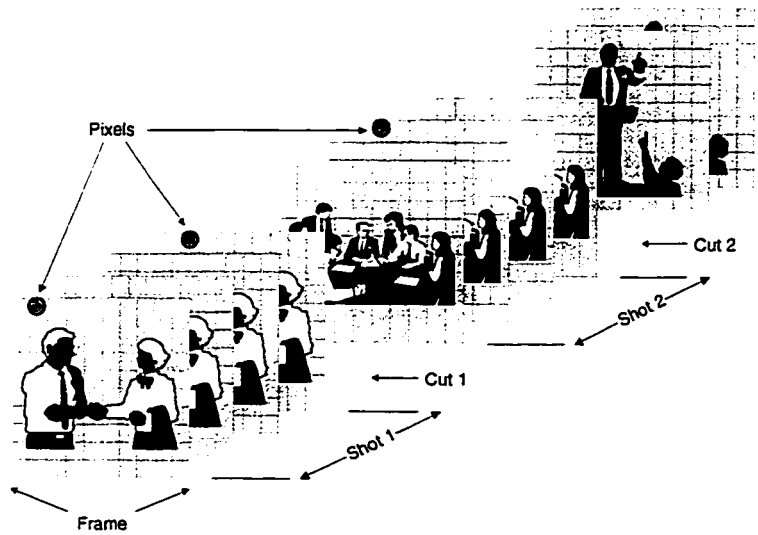


Figure 1: The format of a raw video segment

Cuts can be found automatically by using cut detection algorithms which calculate interframe differences and compare them to certain thresholds. The problem is that interframe differences are not only due to cuts but also due to certain video phenomena such as rapid object motion, strobe lighting, and slow motion [47]. In some cuts, the gradual change of a cut may span more than ten frames which is equivalent to the change introduced by camera movement [19]. Some of these problems have been circumvented by sophisticated cut detection algorithms such the one presented in [17] which was based on the linear prediction of optical flow but, so far, there has been little success in automatic detection of gradual transitions [51].

As will be shown in this thesis, the current version of Video++ depends on video description headers rather than cut detection algorithms to identify and describe the shots in a video segment.

2.3 Image Retrieval

Image retrieval is available in two forms; text-based retrieval and visual content-based retrieval. Most image retrieval systems are text-based.

2.3.1 Text-Based Retrieval

Image databases and visual information systems use a text-based approach to indexing and retrieval. Text description, similar to Video++ description headers, are associated with images in a database. The text description has one of either two forms; key words or free text. Search and retrieval is based on exact or probabilistic matching of the textual information [19,2,10]. The textual information can also be used to build classification hierarchies; such hierarchies may then facilitate navigation and browsing [19,3].

2.3.2 Content-Based Retrieval

The alternative to text-based retrieval is to retrieve images based on their properties i.e. to retrieve visual data using queries based on the visual content of an image. The visual content can be patterns, colors, textures, shapes, and layout. In some cases, visual queries may be easier to formulate than text-based queries. Search is driven by first establishing one or more sample images and then identifying specific features of those sample features which need to match images in the resource being searched. For example, by using algorithms that can match colors and shapes, the search for a red-colored spherical object may be accomplished with a more concise query than the textual equivalent of searching for the keyword "red ball".

The Video Classification Project at the Institute of Systems Science (ISS) of the National University of Singapore (NUS) is an effort to change this situation. It aims to develop an intelligent system that can automatically classify the content of a given video package. The tangible result of this classification will consist of an index structure and a table of contents,

both of which can be stored as part of the package. In this way, a video package will become more like a book to anyone interested in accessing specific information [51].

Content-based retrieval and text-based retrieval need not be mutually exclusive; both may be used simultaneously. Key words can complement visual content retrieval [19]. However, neither can guarantee finding the correct match and so both methods retrieve a number of images that resemble the search criteria; it is up to the user to manually discard the incorrect ones. It is more important that candidates which are likely to be of interest are not excluded than it is that possibly irrelevant candidates be included [19,36].

2.4 Image indexing

The conventional method for indexing images is to associate key words with each image. This is preferably automated since human operators are generally costly, subjective, and unreliable in generating key words [19]. Fully automatic generation of key words, however, is still beyond the current technology of computer vision and is unlikely to be realized in the near future [19].

Many systems depend on human input and feedback when generating the annotation for image data. For example, a system that annotates images based on texture properties would require the user to browse through the images and identify a few samples for the system to use as a reference when the system automatically selects the annotations [38].

A system that supports content-based retrieval is IBM's Ultimedia Manager. Ultimedia runs on OS/2 and uses the DB2 RDBMS. It is expected that this system will be ported to other platforms such as UNIX and Windows [52]. Another example is the Visual Information Management System (VIMSYS) developed as part of the InfoScope project at the University of Michigan [23].

2.5 Video Indexing

The vast majority of video segments that have been acquired sit on a shelf for future uses without indexing [34]. This is due to the fact that indexing requires an operator to view the entire video package and to assign index terms manually to each of its scenes. Such an

approach is currently not feasible; when it comes to the problem of developing new techniques for indexing and searching of video sources, the intuitions that have been acquired through the study of information retrieval do not translate very well into non-text media such as video [51].

When text is indexed, words and phrases are used as index entries for sentences, paragraphs, pages, or documents. This process only selects certain key words or phrases as index terms. Similarly, a video index will require, apart from the entries for scenes used in text, key frames or frame sequences as entries for scenes or stories. Therefore, automated indexing will require the support of tools which can detect and isolate such meaningful segments in any video source. Content analysis can then be performed on individual segments in order to identify appropriate index terms [51].

2.6 Video Compression

Modern video compression techniques reduce the tremendous storage requirements of video. Advanced techniques have achieved video compression ratios of up to 2,000:1. The MPEG compression standard is used in the compression of full-motion video. It uses interframe compression, achieving compression ratios of up to 200:1, by storing only the differences between successive frames. MPEG specifications also include an algorithm for compressing audio data at ratios form 5:1 to 10:1 [20]. The figure below compares the storage requirements of certain media used in multimedia applications.

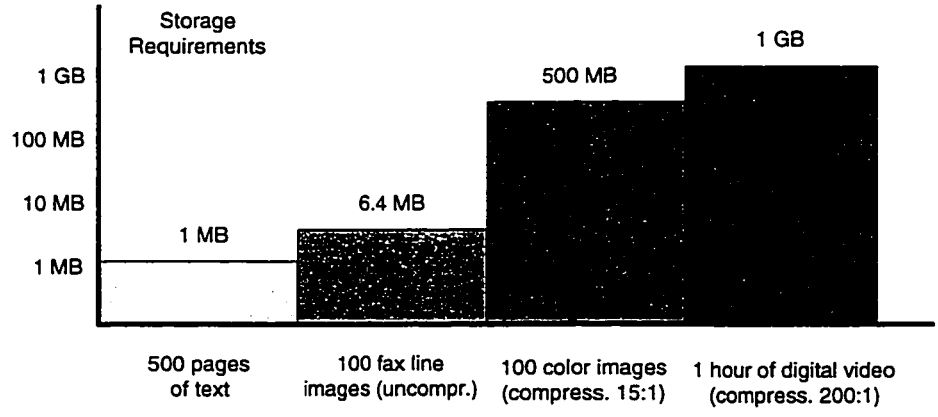


Figure 2: Multimedia Storage Requirements

The design of the MPEG algorithm had to take two conflicting requirements into consideration. On one hand was the requirement for a high compression ratio that was not achievable with *intraframe* coding alone. On the other hand, the requirement for random access to frames was best satisfied by pure intraframe coding. This required a delicate balance between intra- and interframe coding, and between recursive and nonrecursive temporal redundancy [29].

The MPEG video compression algorithm relies on two basic techniques:

- block-based motion compensation for the reduction of the temporal redundancy
- DCT-based[1] compression for the reduction of spatial redundancy

For the block-based motion compensation, the MPEG algorithm uses two interframe coding techniques: predictive and interpolative. Motion-compensated prediction assumes that the current picture can be modeled as a translation of the picture at some previous time. The amplitude and the direction of the displacement need not be the same everywhere in the picture. The motion information is part of the necessary information to recover the picture. Motion-compensated interpolation (also called bidirectional prediction) is used to reconstruct a signal by adding a correction term to a combination of past and future references.

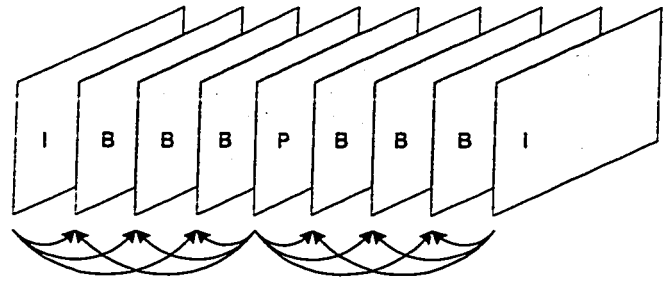


Figure 3: Bidirectional Prediction in MPEG

In the above figure, the three types of pictures that are used by MPEG are illustrated: Intra-Pictures (I), Predicted Pictures (P), and Bidirectional Interpolated Pictures (B). Intra-pictures provide access points for random access but only with moderate compression. Predicted pictures are coded with reference to a past picture (I or P) and will be used as references for

[1] DCT stands for Discrete Cosine Transform

future predicted pictures. Bidirectional interpolated pictures provide the highest amount of compression but require both a past and a future reference for prediction; bidirectional interpolated pictures are never used as a reference. In this particular example, an intracoded picture is inserted every eight frames and the ratio of interpolated pictures to intra- or predicted pictures is three to four.

2.7 The Object-Oriented Paradigm

The object-oriented (OO) paradigm dictates that a system can be viewed in terms of interacting objects. It is the software designer's task to identify these objects and the relationships that bind them. This is in contrast to the functional decomposition approach where a system is decomposed in terms of its functions.

An object is an independent entity that contains functions (also known as operations or member functions) and data [9]. The object provides an interface to the external world. The data of the object can be accessed only if the interface permits it; otherwise, the data is considered private to the object and inaccessible by other objects.

The figure below shows three objects, each one with four interfaces. An object can send messages to the other objects only through those interfaces [27].

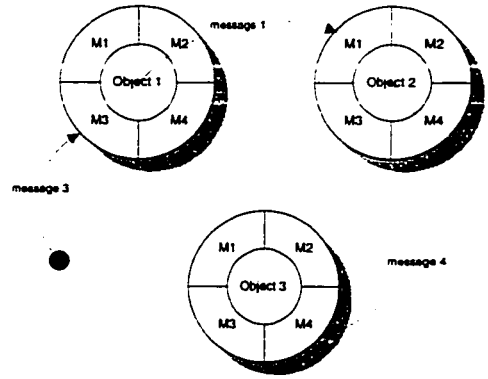


Figure 4: Objects interacting via messages

The three most significant stages in the object-oriented development cycle are analysis, design, and programming.

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain [5].

Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design [5].

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships [5].

Object orientation can, hence, be loosely described as the software modeling and development disciplines that make it easy to construct complex systems from individual components. The main drive to use object-oriented technology is reusability, malleability, and quality of design [9,27].

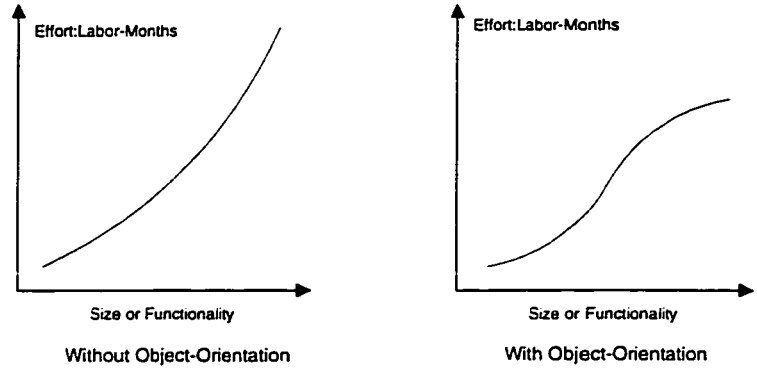


Figure 5: The effect of object-orientation on development effort

The above figure shows that, without object-orientation, the effort to develop a system increases exponentially as a function of size or functionality. With object-orientation, however, linear growth is achieved with increased functionality [27].

2.8 The C++ Programming Language

None of the existing programming languages support the concept of video algebra. To introduce video algebra into an existing programming language, it was necessary to choose a programming language that supports language extensions.

The C++ language was the ideal choice for the following reasons:

- C++ is an object-oriented programming language. An object-oriented language supports encapsulation, inheritance, and polymorphism.
- C++ renders itself easily to the definition of new data types. This made the definition of a new video data type straightforward.
- C++ has a high level of data abstraction which is required for such complex data types as images and video. The *class* construct in C++ proved especially useful.
- C++ allows operator overloading. That means that algebraic expressions dealing with new data types (such as video) look exactly like regular expressions involving regular built-in data types. This produces consistent and neat code and enhances readability. Other popular object-oriented languages, such as Smalltalk and Java, do not support operator overloading [14].
- C++ has become one the most popular programming languages. Java, which is similar to C++, is becoming increasingly more popular but it is unlikely that it will replace C++ in areas that require a high degree of efficiency such as systems programming.
- C++ supports the concept of *manipulators* which provide excellent control over complex input/output operations.

C++ provides language-extensions on a limited basis only. Operator overloading is the process of having one operator take on more than one role. Some operators are already overloaded. The '+" operator is used to add integers, pointers, and real types. Third-party class libraries, such as Rogue Wave's tools.h++ overload the '+' operator even further by using it to add strings and other data types. In C++, only the *predefined* set of operators can be overloaded [31]; in other words, users cannot define new operators such as '#' or '@'.

This means that Video++ is limited to using existing operators which have a predefined meaning i.e. operators which already have some kind of operation associated with them. The ANSI C++ committee is currently conducting research to see whether user-defined operators should be permitted in future versions of C++.

C++ is not an easy language to learn. The figure below indicates the approximate learning curve for becoming proficient in C++ [9].

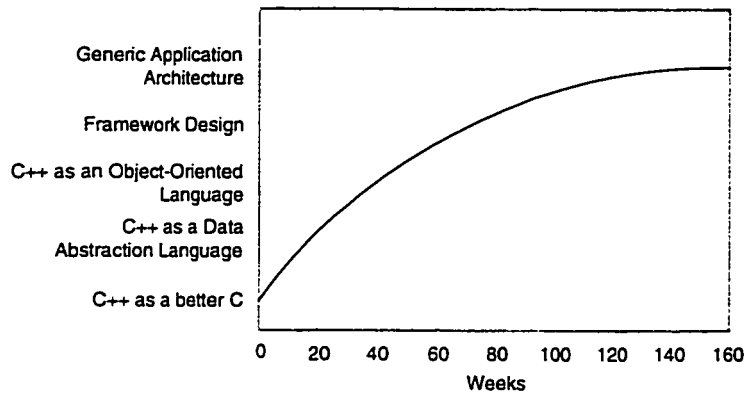


Figure 6: The learning curve for mastering C++

The Video++ framework falls between Framework Design and Generic Application Architecture.

2.9 Summary

This chapter covered the basic concepts that are required to understand the development of Video++. Video has properties that make it more difficult to work with than other types. It was also shown that text-based image retrieval is an acceptable alternative to visual content-based retrieval. A brief introduction to the object-oriented paradigm was presented; the object-oriented paradigm stipulates that systems can be modeled in terms of interacting objects. Object-oriented analysis and design have several advantages over the traditional analysis and design methods. It was shown that the C++ programming language is suitable for the development of Video++.

3. THE MIT ALGEBRAIC VIDEO SYSTEM

The MIT Laboratory for Computer Science has also developed a system for the implementation of a video algebra. The MIT Algebraic Video system (referred to as the MIT AV system from here on) allows users to compose algebraic video representations. It extracts video attribute information and offers a query-based interface for searching, browsing, and playing back relevant video segments [48].

The MIT AV system introduced video algebra as a means of combining and expressing temporal relations, defining the output characteristics of video expressions, and associating descriptive information with these expressions. The MIT team developed algebraic video to let users create video presentations that:

- model nested video structures such as shot, scene, and sequence
- express temporal compositions of video segments
- define output characteristics of video segments
- specify multistream viewing

3.1 Architecture

The design goal of the MIT AV system was to provide a high-level abstraction that models complex information associated with digital video and supports content-based access. The MIT AV system has a graphical user interface for managing a collection of raw video segments and algebraic video nodes; it also includes query and browsing tools. The storage subsystem includes raw, unstructured video, a representation of algebraic video, and indices to support content-based access.

The MIT AV system provides the following functions:

- acquiring video data from external sources such as TV broadcasts or other video collections
- parsing the raw, unstructured video to algebraic video files
- indexing algebraic video nodes

- providing content-based access to the data
- playing back and browsing the video expressions
- composing, reusing, and editing complex video expressions

The MIT AV implementation is built on top of three existing subsystems: the **VuSystem**, the **Semantic File System (SFS)**, and the **World Wide Web (WWW)**. The Semantic File System is used as a storage subsystem with content-based access to data for indexing and retrieving files that represents algebraic video nodes. The VuSystem provides an environment for recording, processing, and playing video. The WWW server provides a graphical interface to the system that includes facilities for querying, navigating, video editing and composing, and invoking the video player. The WWW access module includes static hypertext markup language (HTML) documents and a set of Tcl scripts that dynamically create HTML documents in response to user interaction.

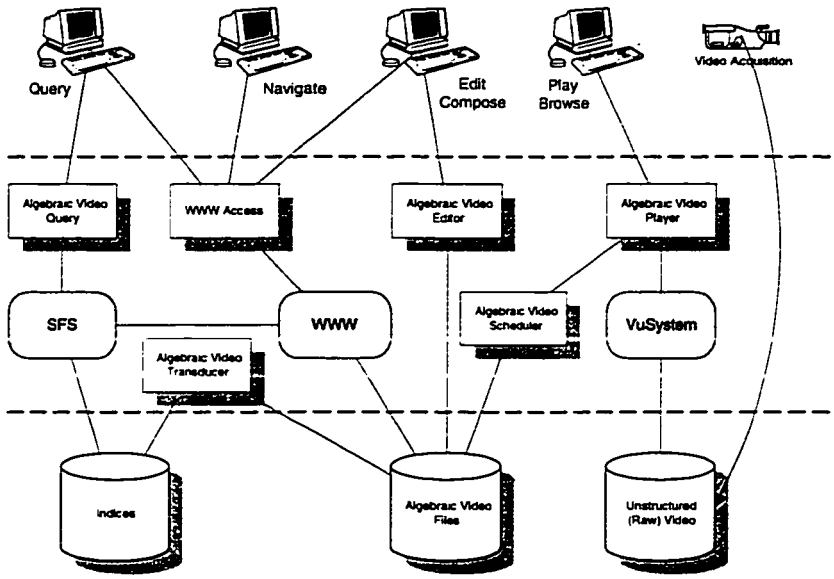


Figure 7: The MIT Algebraic Video System

The AV system indexes the video files to create correspondence between the attributes and the algebraic video nodes. The **Algebraic Video Transducer** in the Semantic File System extracts attributes from the descriptions stored in the algebraic video files. The transducer associates attributes and values with the algebraic video files during the indexing process.

Indexing this information allows efficient querying and retrieval of relevant video segments. Individual video nodes that overlap are indexed separately.

The **Algebraic Video Query** and WWW modules, as well as the user, communicate with the Semantic File System directly via the pathname interface. The Semantic File System interprets a file pathname as an attribute query. It then returns the result in a dynamically created virtual directory that contains the set of matching algebraic video nodes.

Once the user selects a nested hierarchy of algebraic video nodes for playback, the system recursively parses the nodes and compiles a schedule file for each. A schedule file is a Tcl script consisting of window and audio output declarations and a segment activation script. The system implements Playback by dynamically interpreting the schedule files to produce streams of digital video. The streams are transmitted to the VuSystem, which then displays the digital video on the client workstation.

3.2 Content-based Access

The user can associate personalized descriptions with any component of the video presentation. The MIT AV system integrates content-based access to video, allowing the user to:

- associate content information with logical video segments
- provide multiple coexisting views and annotations of the same data
- provide associative access based on the content, structure, and temporal information

3.3 Video Algebraic Operations

The MIT Algebraic System divides the video algebra operations into four categories:

1. Creation
   Defines the construction of video expressions from raw video
2. Composition
   Defines temporal relationships between component video expressions

3. Output

   Defines spatial layout and audio output for component video expressions

4. Description

   Associates content attributes with a video expressions

The table below lists all the algebraic operators available in the MIT AV system.

Operator	Usage	Functions
Creation		
Create	`create` *name begin end*	Creates a presentation from the range within the identified raw video segment
Delay	`delay` *time*	Creates a presentation with empty footage for duration *time*
Composition		
Concatenation	$E_1 \circ E_2$	Defines the presentation where E_2 follows E_1
Union	$E_1 \cup E_2$	Defines the presentation where E_2 follows E_1 and common footage is not repeated
Intersection	$E_1 \cap E_2$	Defines the presentation where only common footage of E_1 and E_2 is played
Difference	$E_1 - E_2$	Defines the presentation where only footage of E_1 that is not E_2 is played
Parallel	$E_1 \parallel E_2$	Defines the presentation where E_1 and E_2 are played concurrently and start simultaneously
Parallel-end	$E_1 \approx E_2$	Defines the presentation where E_1 and E_2 are played concurrently and terminate simultaneously
Conditional	(`test`) ? E_1:E_2: ... :E_n	Defines the presentation where E_1 is played if `test` evaluates to i
Loop	`loop` E_1 *time*	Defines a repetition of video expression E_1 for a duration of *time*
Stretch	`stretch` E_1 *factor*	Sets the duration of the presentation equal to *factor* times duration of E_1 by changing the playback speed of the video expression
Limit	`limit` E_1 *time*	Sets the duration of the presentation equal to the minimum of *time* and the duration of E_1 but the playback speed is not changed
Transition	`transition` E_1 E_2 *type time*	Defines *type* transition effect between expressions E_1 and E_2; *time* defines the duration of the transition effect
Contains	`contains` E_1 *query*	Defines the presentation that contains component expressions of E_1 that match *query*
Output		
Window	`window` E_1 (x_1,y_1) - (x_2,y_2) *priority*	Specifies that E_1 will be displayed with *priority* in the window defined by the bottom-left corner (x_1,y_1) and the top-right corner (x_2,y_2) such that $x_i \in [0,1]$ and $y_i \in [0,1]$
Audio	`audio` E_1 *channel force priority*	Specifies that the audio of E_1 will be output to *channel* with *priority*; if *force* is true, command overrides audio specifications of the component expressions
Description		
Description	`description` E_1 *content*	Specifies that E_1 is described by *content*
Hide-content	`hide-content` E_1	Defines a presentation that hides the content of E_1

Table 2: The MIT Algebraic Video Operations

In the example below, one raw video file is annotated by three overlapping nodes. The union of the three overlapping nodes yields one video stream with no redundancy in the playback.

Example

```
C1 = create Cnn.HeadlineNews.rv 10 30
C2 = create Cnn.HeadlineNews.rv 20 40
C3 = create Cnn.HeadlineNews.rv 32 65

D1 = (description C1 "Anchor speaking")
D2 = (description C2 "Professor Smith")
D3 = (description C3 "Economic Reform")

(D1 ∪ D2 ∪ D3)
```

Another example below illustrates an algebraic video node with nested window specifications.

Example

```
C1 = create hoffa.rv 30:0 50:0

P1 = window C1 (0,0)-(0.5,0.5) 10
P2 = window C1 (0,0.5)-(0.5,1) 20
P3 = window C1 (0.5,0.5)-(1,1) 30
P4 = window C1 (0.5,0)-(1, 0.5) 40
P5 = (P1 || P2 || P3)
P6 = (P1 || P2 || P3 || P4)

(P5 || (window(P5 || (window P6(0.5,0.5)-(1,1) 60))(0.5,0.5)-(1,1) 50))
```

The major similarity between the MIT AV system and Video++ is that both systems apply algebraic expression to video segments. The design goal of Video++, however, differs from that of the MIT AV system in several respects. First, Video++ introduces video as a data type to be used from within a high-level programming language. Video++ uses a set of algebraic operations that is consistent with the algebraic operators of the host programming language. Second, Video++ was designed to simplify the content analysis of video segments. Third, Video++ advocates a distributed approach to video storage. Video segments may be located anywhere on the Internet and will be located during run-time. Fourth, Video++ is a class library that can be used by other applications.

The similarities and differences between the MIT AV system and Video++ are discussed in more detail in Chapter 6.

3.4 Summary

This chapter provided a brief introduction to the MIT AV system. The MIT AV system is built on top of three subsystems; the VuSystem, the Semantic File System, and the World Wide Web. Content-based access is based on annotations. Similar to Video++, the MIT AV system provides a complete set of algebraic operators.

4. DESCRIPTION OF VIDEO++

Automatic extraction of semantic information of general video programs is outside the capability of current machine vision and audio signal analysis technologies [19]. On the other hand "content parsing" may be possible when one has an a priori model of a video's structure based on domain knowledge [19]. The Video++ system defines this 'a priori model' for video and how to use that model for basic video manipulation. This chapter will discuss the theory behind Video++. It will introduce the concepts of video description headers, C++ operator overloading, video dependencies, and tree algebra.

4.1 Video Algebra

Most programming languages support algebraic and logical operators for their built-in data types. Video is not considered a built-in data type and cannot be used in algebraic expression without extending the programming language. This thesis demonstrates that video algebra is feasible and that a set of algebraic and logical operations for video can be defined.

When parsing algebraic expressions, most compilers will hide implementation-specific details from the programmer. For example, floating point types can have complex internal representations that do not appear in high-level programs as shown below:

```
int y = 5;
float x, z = 3.14159

x = y + z;
```

The C++ compiler will parse the last statement, convert all values to the floating point type, and invoke the addition algorithm for floating point types.

The last statement in the code above illustrates the concept of *overloaded* operators. The "+" operator works for pointers, integers, and floating point numbers. Programming languages that do not support overloaded operators would have to use four different symbols for the addition of floating and integer types.

The table below shows the four types of addition for integers and floating types:

(int)	$+_1$	(int)
(int)	$+_2$	(float)
(float)	$+_3$	(int)
(float)	$+_4$	(float)

The same would hold true for other operations. The table below lists the algebraic and logical operators that constitute a video algebra.

Operation	Symbol	Logical Equivalent	Logical Symbol	C++ Expression
Addition	+			(a + b)
Subtraction	-			(a - b)
Union	$\cup$	OR	$\times$	(a | b)
Intersection	$\cap$	AND	$\bullet$	(a & b)
Unisection		XOR	$\otimes$	(a ^ b)
Subset	$\subset$			(a < b)
Superset	$\supset$			(a > b)
Equality	=			(a == b)
Compression				(a--)
Decompression				(a++)
Subscript				a[n]
Range				a(n, m)
Search				a("context")

Table 3: Video algebra operations

Most of these algebraic operations are straightforward and have equivalents in algebra, boolean logic, and set theory. Operators such as compression and decompression are unique to video. It will also be shown later in this thesis that addition, subtraction, union, intersection, and unisection can also be executed by using the following complimentary algebraic operators, respectively: +=, -=, |=, &=, and ^=.

A video segment is analogous to a set in set theory; the basic elements of a video segment are frames. There is, however, one fundamental difference. Video has a temporal dimension and the temporal sequence of frames must be preserved. For some binary algebraic operations, it is impossible to preserve the temporal order of both operands and the temporal order of *one* of the operands is ignored. In this case, it is always the first operand that dictates the temporal order of the result. For example, let us assume that two video segments have two short

sequences in common. Giving each sequence an alphabetic tag, the first video segment can be represented by abcdevgh and the second video segment by pqrsvtucw. The c and v sequence are common to both and have been underlined. These two sequences have different temporal orders in the original video segments; if the result of an algebraic operation includes both sequences, the temporal order of the first operand is chosen. Hence, the intersection operation, which returns all common sequences of its two operands, would return cv and not vc.

In set theory, certain operations can result in empty sets. Similarly, algebraic expression can have a result of zero. The equivalent in video algebra is the **null video** which is a video of zero number of frames. The null video may still have a title and other attributes and may still be used in algebraic expressions. Similarly video algebra expressions that deal with frames (such as the subscript operator) must return a **null frame** if none of the frames in the video segment are chosen.

The video algebra operators will now be discussed in more detail. For illustration purposes we will use the two sample video segments shown below.

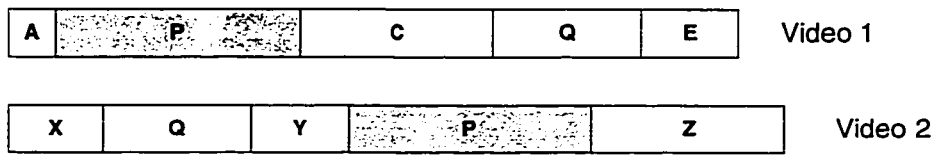


Figure 8: Two sample video segments

Each video segment consists of one or more "frame sequences". Frame sequences, which consist of an arbitrary number of related shots and frames, are sometimes called *episodes* [48]. The second video segment is slightly longer than the first. Also, the two video segments have two frame sequences in common; sequence P and sequence Q. Two frame sequences are considered equal if they have exactly the same frames, in exactly the same temporal order. This means that all the frames that occur in sequence P and Q in Video 1 can also be found in Video 2. The temporal order of these two common frame sequences is not the same, however; in the second video segment, sequence P occurs after sequence Q.

4.1.1 Addition

Type	Binary
Commutative	No
Associative	Yes
Result	video

Description

The addition operation takes two video segments and concatenates them. Any common footage is duplicated in the result. Addition of video segments is not commutative i.e. (A+B) ≠ (B+A). The reason is that video has temporal properties that are not respected by the commutative law. Hence, a video segment where sequence A occurs before sequence B is not considered equal to a video segment where sequence A occurs after sequence B. Addition of video, however, is associative, i.e. (A+B)+C = A+(B+C), because the temporal order is not affected. Addition is also known as *concatenation.*

Example

The addition of the two sample video segments introduced above gives the following result:

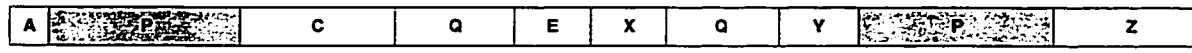


Figure 9: Addition of the two video operands

4.1.2 Subtraction

Type	Binary
Commutative	No
Associative	No
Result	video

Description

The subtraction operation removes the second video segment from the first. If the second video segment contains footage that is present in the first video segment the result is the first

video segment excluding that common footage. If the two video segments do not share any common footage the result is the first video segment. Subtraction is also known as *difference*.

Example

The subtraction of the second video segment from the first video segment would give the following result:

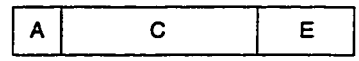


Figure 10: Subtraction of video 2 from video 1

The subtraction of the first video segment from the second video segment would give the following result:

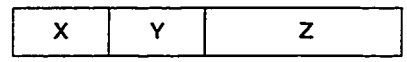


Figure 11: Subtraction of video 1 from video 2

4.1.3 Union

Type	Binary
Commutative	No
Associative	Yes
Result	video

Description

The union operation is similar to addition except that common footage is not duplicated in the result. Union is also known as *OR*, *logical sum*, and *disjunction*.

Example

The figure below shows the union of the first video segment with the second video segment.

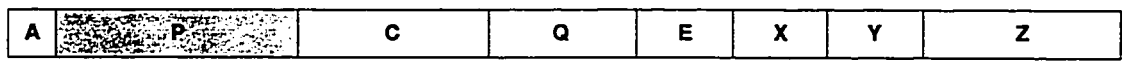


Figure 12: The union of video1 and video 2

4.1.4 *Intersection*

Type	Binary
Commutative	No
Associative	Yes
Result	video

Description

The result of the intersection operation is footage that is common to both video segments. If there is no common footage the result is a null video. Intersection is also known as *AND*, *logical product*, and *conjunction.*

Example

The figure below shows the intersection of the first video segment and the second video segment.

Figure 13: The intersection of video 1 and video 2

To illustrate that intersection is not commutative, the figure below shows the intersection of the second video segment and the first video segment.

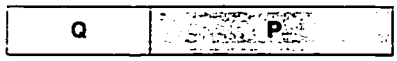


Figure 14: The intersection of video 2 and video 1

4.1.5 *Unisection*

Type	Binary
Commutative	No
Associative	Yes
Result	video

Description

Unisection is the sum of the footage that is not common to any of the operands. Unisection is the equivalent of (Union - Intersection) and its logical equivalent is the XOR operation.

Example

The figure below shows the unisection of the first video segment and the second video segment.

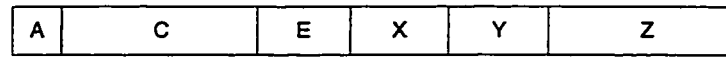


Figure 15: The unisection of video 1 and video 2

4.1.6 *Equality*

Type	Binary
Result	Boolean

Description

Two video segments are considered equal if they have the same sequence of frames. In Video++, video segments are considered equal if they have the same content i.e. the video segments need not match at the bit level if they have identical text descriptions. The result of the equality operation is either True or False. The temporal order must be the same for both operands.

4.1.7 *Inequality*

Type	Binary
Result	Boolean

Description

Two video segments are considered unequal if either their frames or their temporal orders do not match. In Video++, video segments are considered unequal if they have do not have the same content. The result of the inequality operation is either True or False.

Example

The two video segments in the figure below are not equal; they do not share the same temporal order.

Figure 16: Two video segments which are not equal

4.1.8 *Subset*

Type	Binary
Result	Boolean

Description

A video segment is considered a subset of another if all its frames are contained in the other. The result is either True or False. The temporal order of the subset must be the same as the original video segment.

Example

In the figure below the first video segment is a subset of the second video segment.

Figure 17: The subset of a video segment

4.1.9 Superset

Type	Binary
Result	Boolean

Description

A video segment is considered a superset of another if it contains all the frames of the other. The result is either True or False. The temporal order of the original video segment must be the same as in the superset.

Example

In the figure below the first video segment is a superset of the second video segment.

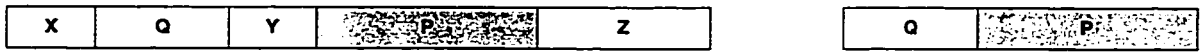


Figure 18: The superset of a video segment

4.1.10 Compression

Type	Unary
Result	Video

Description

Compression does not have any logical or arithmetic equivalent; it is unique to video algebra. It takes as an operand an uncompressed video segment and compresses it. If the video segment is already compressed, no action is taken. Compression is also known as *reduction*.

4.1.11 *Decompression*

Type	Unary
Result	Video

Description

Decompression does not have any logical or arithmetic equivalent; it is unique to video algebra. It takes as an operand a compressed video segment and decompresses it. If the video segment is already compressed no action is taken. Decompression is also known as *expansion*.

4.1.12 *Subscript*

Type	N/A
Result	Frame

Description

The subscript operator returns a frame as specified by the subscript **index**. If the subscript index is out of range. the result is a null frame. In set theory, the subscript operation is analogous to specifying an element in a given set.

Example

The figure below shows the extraction of a single frame from a video segment using the subscript operator. The frame was extracted from the frame sequence P. If the video segment had been compressed, it would have had to be decompressed before applying the subscript operator.

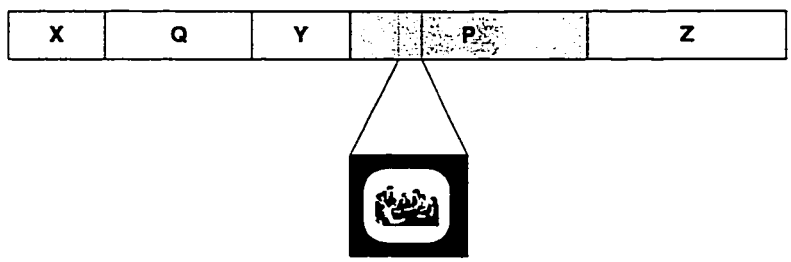


Figure 19: The subscript operation

4.1.13 Range

Type	N/A
Result	Video

Description

The range operator returns a range of frames as specified by the range indices. If one of the indices is out of range the result is a null video. In set theory the range operation is analogous to specifying a subset in a given set.

Example

The figure below shows the extraction of a range of frames (3 frames) from a video segment using the range operator. All three frames were extracted from the frame sequence Q. If the video segment had been compressed, it would have had to be decompressed before applying the range operator.

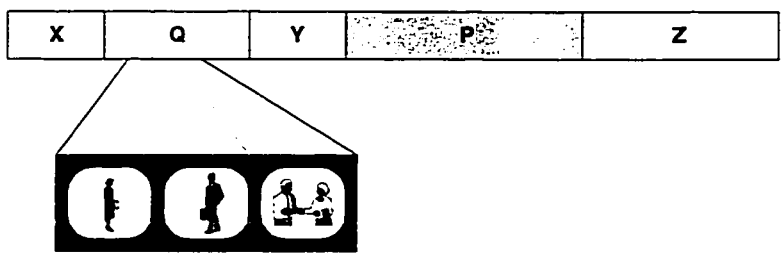


Figure 20: The range operation

4.1.14 Search

Type	N/A
Result	Video

Description

The search operator is similar to the subscript operator but instead of using an index to locate the frame it performs a context-sensitive search using a search keyword. This operator is by far the most complex and compute-intensive since it may involve complex pattern

recognition algorithms. Later sections will explain how the search operator is implemented in Video++ using text-based content search.

4.2 Video++

Video++ is a class library written in C++. The objective of Video++ is to make video as easy to use as the built-in data types. The corresponding advantages are:

- Consistency
- Neatness
- Ease of understanding
- Ease of usage
- Ability to build complex systems
- Extensibility

Video++ implements all of the algebraic operations in a simple and straight-forward manner by hiding the internal complexities of the video data from the developer.

We define the concept of a video description header to successfully implement the algebraic operations. Briefly, video description headers have two main functions. First, they provide textual information that allows Video++ to extract and interpret the contents of video. Second, video description headers contain *links* to other video segments; these links facilitate the implementation of a video algebra.

Video description headers are meant to complement the visual content already present in the video segment. The information provided through textual descriptions is usually not as comprehensive as the information provided through the visual content; on the other hand, automatically extracting all the visual content from a video is very difficult using current technology. Although there have been significant advances in automatic feature extraction and pattern recognition it is unlikely they can capture the cultural, historical, and personal information that is usually present in video. Textual information, on the other hand, can

convey abstract things like mood, emotions, background, etc. through proper descriptive terms.

The video description headers are written (or generated) in a language called VML. The size of the header can range anywhere from zero to several kilobytes. Video++ makes extensive use of these headers and its algebraic operations are restricted if these headers are absent. In addition to reading these headers Video++ also updates them, depending on the algebraic operation being performed on the video segment.

Video++ distinguishes between **intra-video** sequences and **inter-video** sequences. An intra-video sequence contains frames that belong to the same physical video segment. Inter-video sequences, on the other hand, contain frames that belong to different physical video segments. This allows for reusability. For example, a documentary about the civil-rights movement in the 1960s need not physically contain actual footage from that time; instead it may point to various segments located at remote locations such as the Library of Congress and the Smithsonian Institute. Similar concepts are already used in programming languages such as the C/C++ language (which uses pointers) and HTML (which uses hyperlinks). It will be shown later that algebraic expressions can be applied only to inter-video sequences.

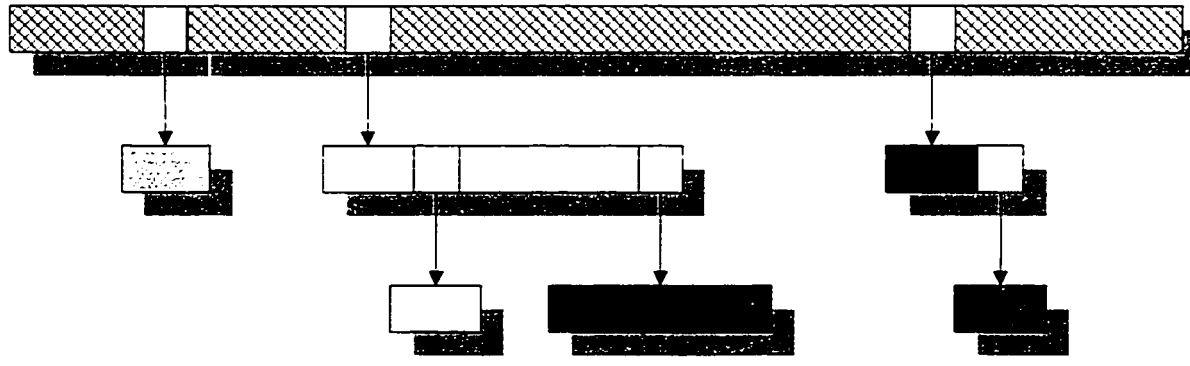


Figure 21: A video segment with inter-video frame sequences

It may be argued that having a central database of video description headers is better than having each video stream carry its own header. However, the Internet, and specifically the World Wide Web, has proven that decentralization has its advantages. Maintaining one large database would be too cumbersome. Every video segments maintains its own header locally. This is analogous to maintaining home pages on the World Wide Web.

4.3 The Video Markup Language (VML)

The description of video content is more complicated than the description of static image content. What makes video different from static images is the temporal factor. In describing a single image, attention tends to be focused on the objects in that image and how they are spatially related. However, the time dimension introduces the concept of an event [19]. Events, like object in static images, also need to be described. This section will introduce VML, a video description language that encompasses object, events, and other information such as audio.

Every video segment should have a header that contains textual information about the video content. However, if it is assumed that multiple users will be examining this text and extract information from it, then it is useful to impose a syntax on how that text will be written [19]. To implement this syntax, the Video++ system introduces a new markup language called VML. VML is similar to the Standard Generalized Markup Language (SGML) which is considered as the superset of many markup languages. SGML is the ISO standard for document description. It is designed to enable text interchange and is intended for use in the publishing field, but can also be applied in the office and engineering areas. SGML documents have a rigorous structure that may be analyzed by computers and easily understood by humans [24]. Other well known languages based on (or associated with) SGML are HTML (the HyperText Markup Language), SPDL (the Standard Page Description Language), SMDL (the Standard Music Description Language), and HyTime (the Hypermedia/Time-based structuring language) [44].

When SGML was designed it had to meet many criteria. Specifically, it had to be independent of system, devices, languages, and applications [44]. A language based on SGML, therefore, would be also suitable for the description of video in Video++.

VML, like any SGML-based language, is always in text format (i.e. printable ASCII characters). This has the following advantages:

- VML files can be edited with any text editor.
- VML can be easily read and understood.
- The ASCII format is portable across heterogeneous systems.

A video description header is divided into VML sections. Any section in the header is optional. The header itself can be either empty or absent. This was done for compatibility purposes; Video++ would still be able to use existing video segments that do not have video description headers.

4.3.1 HyTime

In the HyTime standard, Hytime is defined as follows. "HyTime provides a standard way to link to anything, anywhere, at any time." [22]. A more formal definition is: HyTime is a proposed standard language for representing the structure of multimedia, hypertext, hypermedia, time- and space-based documents. HyTime is a collection of abstract semantic constructs associated with syntactic conventions. It allows hyperdocument interoperability to the maximum extent possible without standardizing multimedia objects, their notations, their modifiers, the effects of those modifiers on them, or the semantics of link types, and without requiring existing documents to be recast in order to make their contents linkable by HyTime documents. HyTime-compliant documents can allow HyTime-cognizant software to browse, render, format, and query them even if that software is not able to understand or render its multimedia objects. If the notation of an object is uninterpretable because no interpreting system is locally available for it, the rendering can still incorporate some form of blankness - darkness, silence, or an appropriate error message - so that the space and time relationships of the rendered and unrenderable objects is preserved [35].

HyTime, like VML, provides a mechanism for specifying hyperlinks and scheduling multimedia information in time and space. HyTime, however, is intended for final, formatted, multimedia documents and lacks the mechanisms for content-based access or editing and annotating the multimedia data [48].

4.3.2 VML syntax

Like most languages the video markup language has a syntax. VML syntax is similar to the SGML syntax but is simpler and smaller. In SGML-based languages, the symbols < , > , <! , and </ are called **delimiters** since they delimit the markup. These delimiters are used to construct tags (such as <FRAME>) which markup the logical parts, or **elements**. [24]

4.3.2.1 Symbols

To summarize, the following symbols are used in VML:

Symbol	Name	Function
,	comma	to separate sequences of frames
..	range	to indicate a range
[ ]	brackets	to identify a frame or a set of frames
<	start-tag open delimiter	to identify start of tag
</	end-tag open delimiter	to identify start of end-tag
>	tag close delimiter	to identify end of tag
<>	angle brackets	general markup delimiters
<!--	comment start delimiter	to start a comment
-->	comment end delimiter	to end a comment
&	entity reference	to identify variables
.	point	used for inter-frame stream identification

Table 4: VML Symbols

4.3.2.2 Comments

As in most programming languages, comments must be available to improve readability and provide explanations. A comment field must be start with the "<!--" character sequence and end with the "-->" character sequence.

Example

```
<!--  This is a sample comment -->
<TITLE> Jurassic Park </TITLE>
```

4.3.2.3 Keywords and Tags

The following are valid VML keywords: home, frame, shot. The following are valid VML tags: INCLUDE, DESCRIPTION, SHOT, TITLE, SEQUENCE. None of these should be redefined by the user.

4.3.2.4 Tags

Blocks in VML start with an <identifier> tag and end with an </identifier> tag where the identifier can be any one of the valid VML tags. The valid tags are TITLE, FRAME, SHOT, and INCLUDE.

Example

```
<TITLE> The Deerhunter </TITLE>

<INCLUDE> part1 /usr/someuser/videos/deer_part1.v++ </INCLUDE>
<INCLUDE> part2 /usr/someuser/videos/deer_part2.v++ </INCLUDE>

<DESCRIPTION>

  &part1.frame[0..2455] = "Robert DeNiro and Christopher Walken working
  in a factory"

  &part2.frame[30000..] = "Main characters attend dinner"

</DESCRIPTION>
```

4.3.2.5 The INCLUDE tag

The INCLUDE section serves two purposes; it allows the inclusion of other video segments and it allows the declaration of variables. These variables can then be used in other parts of the video description header by applying the reference operator (&).

In the example below, the INCLUDE section creates and uses the variables `first`, `second`, and `third`.

Example

```
<TITLE> The 8 p.m. News </TITLE>

<INCLUDE> first  /usr/someuser/videos/headlines.v++   </INCLUDE>
<INCLUDE> second /usr/someuser/videos/mexico_1.v++    </INCLUDE>
<INCLUDE> third  /usr/someuser/videos/auto_strike.v++ </INCLUDE>

<DESCRIPTION>

  &first.[0..] = "Peter Jennings introduces himself"
  &second.frame[3122..5244] = "Talks about the crisis in Mexico"
  &second.frame[5245..7880] = "Shows footage of Raul Salinas' estate"
  &third.[0..] = "Talks about Auto workers strike at GM plant"

</DESCRIPTION>
```

In VML, as in SGML, there is no need to place an equal sign between a variable that is being defined and its value. The syntax `frame.[0..]` means the entire frame sequence, starting from frame 0 until the end.

4.3.2.6 The DESCRIPTION tag

This section provides the bulk of the descriptive information for video segments. Frames are identified using the following syntax:

Frame[<exp>, <exp>, ...]

where <exp> is any algebraic expression that resolves to an integer or a set of integers. Frame[0] is always the first frame in a video segment. Commas separate sequences of frames. The '..' operator indicates a continuous sequence of frames. The following table illustrates some examples.

Frame	Description
[n]	is the n^{th} frame
[n..]	all frames starting from the n^{th} frame
[n..m]	all frames starting at the n^{th} frame and ending at the m^{th} frame
[n, m]	two frames; the n^{th} frame and the m^{th} frame
[n..m, p, q..]	denotes three sets of frames; one ranging from frame n to frame m, the single frame p, and all frames from frame q to the end.

Table 5: Frame expressions

For inter-video sequences, frames must be identified by using the corresponding video variable.

Example

```
<TITLE> The Wizard of Oz </TITLE>

<INCLUDE> promotion /usr/someuser/videos/disney.v++    </INCLUDE>
<INCLUDE> home /usr/someuser/videos/wizardofoz.v++     </INCLUDE>

<DESCRIPTION>

  &promotion = "Disney promotion"
  &home[3122..5244] = "Toto exposes the wizard"

</DESCRIPTION>
```

4.3.2.7 The SHOT tag

The SHOT tag is used to identify shots in the video segment. Once shots have been identified, they can be used in the DESCRIPTION section just like frames. In the example below, two shots are described in one statement. Each shot may contain one or more frames.

Example

```
<TITLE> The 8 p.m. News </TITLE>

<INCLUDE> home /usr/some user/videos/news.v++ </INCLUDE>

<SHOT> home.shot[1] home[0..2376] </SHOT>
<SHOT> home.shot[2] home[3277..8776] </SHOT>

<DESCRIPTION>

  home.shot[1] = "The signing of the peace treaty"

  <!-- which is equivalent to -->
  home.frame[0..2376] = "The signing of the peace treaty"

</DESCRIPTION>
```

In the above example, two shots are identified and described. The first shot, `home.shot[1]`, starts at frame 0 and ends at frame 2376. The second shot, `home.shot[2]`, starts at frame 3277 and ends at frame 8776. As in the INCLUDE section, there is no need to provide an equal sign (=) in the SHOT section; VML properly interprets the white space between the variable and its value.

4.3.3 Alternative description languages

Another method of describing video is to use an event-based text description. This is similar to a schedule which accounts for all events with respect to what they are and when they occur. Below is a graphic representation of such a schedule; this kind of representation is also known as a stratagraph. This approach to video annotation is known as the Stratification method [1].

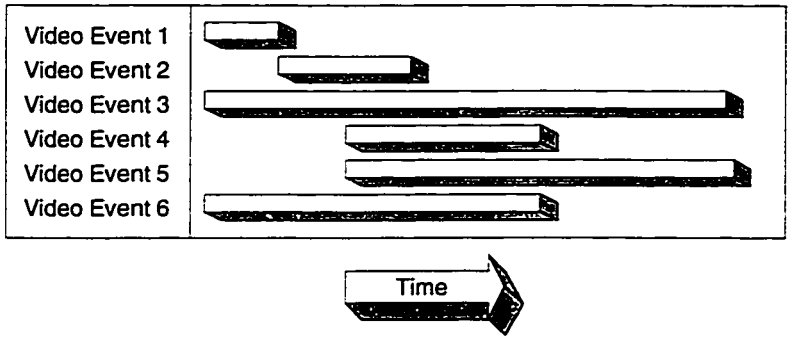


Figure 22: A stratagraph

An event is a sequence of related actions. For example, in a police drama a car chase may be considered an event. Event-based descriptions differ from Video++'s approach in that they describe events and not individual frames or shots. This may have the advantage of giving more coherence to the plot of the video segment. On the other hand, event-based description have a tendency to span several shots (if not the entire segment) hence making the text-based search for individual frames and shots more difficult.

It is possible to extend the VML language to support events. Video description headers would then contain an event section in addition to the other sections which describe individual frames and shots.

4.4 The Video Description Header (VDH)

Video is a medium that cannot be easily annotated [32]. Marking and logging of video must take place on other media, such as paper or computer records. There is no easy way to write "in the margins" of a videotape, so that comments on an edit decision or opinions on the subject matter can easily be passed on to others [32]. In Video++, video description headers are used to annotate and describe the contents of video segments. Ideally, every video segment should have a video description header to describe its contents. Video segments which do not have a video description header are referred to as *orphaned* video segments.

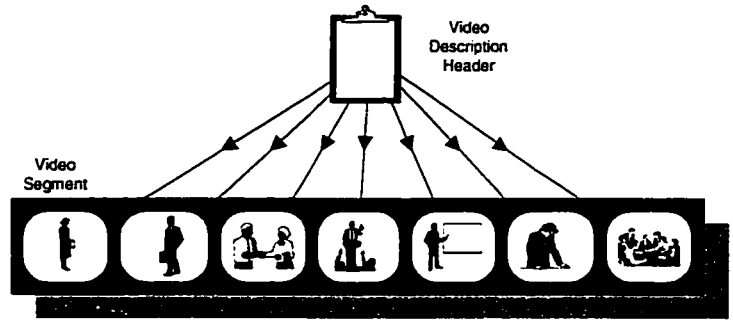


Figure 23: The duality of video

In Video++, there are two kinds of video description headers, home video headers and compound video headers.

Home video headers describe **at most** one contiguous video segment known as the 'home' video segment. The home video header and its corresponding home video segment must reside in the same directory. Future versions of Video++ may require the home video header to be physically attached to the video segment. Compound video headers describe one or more video segments. These headers are not required to reside in the same directory as the video segments. The most important feature of compound video headers is that they can link inter-video sequences.

Home video description headers, which contain the DESCRIPTION section, must be written manually; it would be difficult to generate a DESCRIPTION section from the contents of the video segment without some human intervention. Compound headers, however, could be automatically generated during the execution of video algebraic expressions.

The primary purpose of video description headers is to provide descriptive information for the video segment. If the video segment is moved from its directory, it is imperative that the corresponding link in the video description header be updated. Video++ does not have the capability to automatically monitor the movement of video segments. Device drivers, however, do have the power to monitor the I/O operations of the operating system. Device drivers are low-level extensions to the operating system that may handle tasks such as memory management, disk I/O, and other low-level functions [16]. Device drivers in UNIX are written in a high-level language like C or C++ and are loaded when the operating system

boots. Some operating systems, such as SUN Solaris and DEC VMS, can load and unload device drivers dynamically after the operating system has booted.

Video description headers are written in VML. As such, video description headers can be considered to consist of logical parts, or elements. These logical parts are:

- **INCLUDE** sections
- **TITLE** section
- **SHOT** sections
- **DESCRIPTION** sections

These section provide information on video segments such as title, location, length (in minutes), rating, classification, subject, and content description. All of them may be used for searching, playback filtering, and other user-defined functions. More sections may be added in the future. For example, a **CREDITS** section that lists the names of the actors, producers, etc. could provide useful information about the video segment. Cameos are special takes that usually point out some interesting feature in a movie such as a short appearance by a well-known character. Adding a **CAMEO** section is another option .

A video description header would therefore have the following skeletal form:

```
<INCLUDE> ... </INLUDE>
<TITLE> ... </TITLE>
<SHOT> ... </SHOT>
<SEQUENCE> ... </SEQUENCE>
<DESCRIPTION> ... </DESCRIPTION>
```

4.4.1.1 The INCLUDE sections

The INCLUDE sections inform Video++ that the current (home) video segment includes frames (or sequences of frames) from other video segments. These video segments are referred to as *linked* video segments. The order of the INCLUDE sections is important. Video description headers that have one or more INCLUDE sections are called dependent or compound video headers. Video segments whose video description headers do not have an include section are known as independent or home video segments.

A video description header may have as many INCLUDE sections as required. However, linking in remote video segments involves file system read/write access, which is slow compared to operations executed in memory.

4.4.1.2 The TITLE section

The title should be the file name of the video segment. If the video segment is a movie then the name of the movie should be used for the title. If the video stream represents some amateur footage taken by a hobbyist then an appropriate title should be chosen. Most content-based searches look at the TITLE field before scanning the DESCRIPTION section.

A video description header is allowed to have at most one TITLE section.

4.4.1.3 The SEQUENCE section

Intra-video and inter-video sequences are identified in this section of the video description header. The sequence section informs Video++ of the temporal order in which the frames should be played. A video description header is permitted to have at most one SEQUENCE section.

Inter-video sequences can be used only if the corresponding video segments were declared in the INCLUDE sections of the video description header. In the example below, the two variables, `video1` and `video2`, were declared in the INCLUDE section prior to using them in the SEQUENCE section.

Example

```
<INCLUDE> video1 /usr/luke/mpeg/bus.v++ </INCLUDE>
<INCLUDE> video2 /ftp/video/pingpong.v++ </INCLUDE>

<SEQUENCE>
   &home.frame[0..99];
   &video1.frame[0..199];
   &video2;
   &home.frame[100..];
</SEQUENCE>
```

Video++ interprets this header is as follows:

1. Playback the first 100 frames of the home video segment. The home video segment need not be explicitly mentioned in the INCLUDE section.

2. Playback the first 200 frames of the bus.v++ video segment. This video segment is found at a remote location.
3. Playback all of the pingpong.v++ video segment
4. Playback the rest of the home video segment starting at frame 100.

The SEQUENCE section may be used to filter out specific sequences. If for example frames 2345 up to 3456 contain violent scenes they may be excluded by ignoring them in the SEQUENCE section. A compound video header that has a SEQUENCE section may include another video description header that already has a SEQUENCE section; if there is a conflict in sequences then the sequence section in the *parent* video header overrides the SEQUENCE section in the *child* video header. Furthermore, if there is a conflict between the INCLUDE section and the SEQUENCE section, the sequence section overrides the include section. For example, if the INCLUDE section provides links to three remote video segments, the SEQUENCE section can dictate which of these video segments is to be excluded. If, however, the entire SEQUENCE section is missing, then all the video segments linked by the INCLUDE section are included in the final playback.

Individual frames cannot be accessed directly in MPEG-compressed video. The reason is that frames are not compressed individually as in MotionJPEG. To locate the n^{th} frame one must first locate the previous 'I' frame [19]. Hence an operation like `my_video[164]`, which accesses the 164^{th} frame, may take longer for an MPEG-compressed video than for a raw video segment.

4.4.1.4 The SHOT sections

Textual descriptions may also be applied to shots. Shots may be identified either manually, by inspecting the video segment, or automatically, by a cut detection algorithm. In either case, all shots should be included in this section if they are to be used in the DESCRIPTION section. If a shot is used in the DESCRIPTION section without having been identified in the SHOT sections, the description is ignored and Video++ issues a warning.

4.4.1.5 The DESCRIPTION section

The DESCRIPTION section is the most important section in a video description header. The entire video content is presented in the DESCRIPTION section in narrative form. The DESCRIPTION section may describe an individual frame, a range of frames, an individual shot, or a range of shots.

Example

```
<INCLUDE> video1 /usr/someuser/videos/video1.v++ </INCLUDE>
<INCLUDE> video2 /usr/someuser/videos/video1.v++ </INCLUDE>

<SHOT> video1.shot[1] video1.frame[0..2365] </SHOT>

<DESCRIPTION>

  video1.frame[23..654] "some description"
  video2.frame[23..654] "further description"
  video1.shot[1] "describe this shot"

</DESCRIPTION>
```

A video description header may have as many DESCRIPTION sections as required.

Video++ is not the only system that stresses the importance of textual information. There are many tools that make use of textual information when other methods fail. In the Unified Method [6] , which is a methodology for object-oriented analysis and design, Grady Booch and James Rumbaugh stress that some modeling information is more conveniently manipulated using text. The text would act as a semantic backplane that provides textual annotations to supplement the models. Rumbaugh and Booch also state that one or more tool may process this textual information for interpretation following agreed-upon semantics. In this thesis, those semantic backplanes are the video description headers. The tool is Video++.

4.4.2 VDH Storage

A fundamental problem is to determine where and how the video description headers should be stored. The following options are possible:

- To store the header in a separate file in the same directory as the video segment. The header will have the same filename as the video stream but with the .v++ extension. For example if the video stream is **pingpong.mv2** then the header

name would be **pingpong.mv2.v++**. The problem with this approach is that the header and the actual video stream may get separated from each other.

- To concatenate the header with the video stream to form one single file. It will then be up to Video++ to extract the header from the actual video segment. The concatenation and extraction are straightforward and can be implemented with simple C/C++ file routines. The problem with this approach is that the new file would be readable only by Video++ and not other video-processing software.
- To integrate the header with the video stream. Every video segment would have a predetermined number of reserved fields for the header. The problem with this approach is that there would be a need for an universal agreement on the video file structure.

In the current release of Video++, the first option has been chosen because it is the simplest and because it allows the immediate use of remote video segments. The three options mentioned above are not necessarily mutually exclusive and future versions of Video++ may support all three of them.

4.4.3 Video Distribution

Unlike many conventional video retrieval systems, Video++ does not attempt to store all video segments in one location. Storing a large number of videos in a database is not feasible with current database technology [19]. Instead, Video++ locates video segments at remote locations on the Internet. Video++ uses a combination of IP address and directory path name to locate a particular video segment and its header.

4.5 Further VDH Applications

In the Video++ system, video description headers have two main functions; they provide links to remote video segments and they provide text-based content descriptions. Video description headers, however, can have practical applications outside the context of Video++.

4.5.1 The CCIR Universal Header/Descriptor

The proliferating variety of public and proprietary video formats (PAL, NTSC, SECAM, DVI,CD-I, CD-TV, JPEG, MPEG, analog and digital HDTV proposals) suggests the need for a universal video descriptor [53]. The descriptor would allow any video segment to be interpreted on any device (television, computer, telephone, or hybrid device) independently of the underlying data embedded in the descriptor. The descriptor is a high-level wrapper that allows at a minimum an identification of the type of video that has been received.

The CCIR descriptor is a data structure consisting of header, subheader, and content. The fixed-length header identifies the format with full error correction, specifies the size of the enclosed video segment and can be used for synchronization purposes. The variable-length subheader is scalable; it can describe an arbitrary amount of detail about the video segment. Such a subheader could contain a wealth of information about the content. The following are examples of information embedded in the subheader: processing history, copyright information, software algorithms for decoding, and compression type [30].

Hence, a universal header and digital representation for video could be an essential mechanism to permit global exchange of content. In that respect, Video++ video description headers are very similar to the CCIR universal headers and could be modified to become CCIR compliant..

4.5.2 Audio Support

VML may be extended to describe, in text form, the audio part of the video segment. The audio part of a video segment may include the background music, background noise, voices, and the lines of individual actors. Video description headers would either support a new AUDIO section or the audio part would be integrated within an existing section. Only a few video segments that are available on the Internet have audio capabilities and computer speakers are not as ubiquitous as other peripherals. For this reason, the support of audio in Video++ has been left as a future enhancement.

4.5.3 *Video Indexing*

Video is, by nature, unstructured. A video segment is basically a series of camera shots, none of which need exhibit any meaningful relationships among each other [19]. Hence, finding suitable keys for video indexing and retrieval for raw video segments is difficult, if not impossible. Video description headers, in addition to providing content information, are useful for video indexing and retrieval.

4.5.4 *Video-On-Demand Databases*

An important application of video databases is video-on-demand systems. In the U.S. some cable and telephone companies are already providing VOD (or Near VOD) services to its customers who can pick a movie by querying the database [11]. One example is the Berkeley Distributed VOD System which provides three basic types of indexes to support user queries: bibliographic, structural, and content. Bibliographic indexes would include the video's title, an abstract, keyword classification of subject and genre, and the sort of information displayed in the opening and closing credits. Structural indexes are based on the temporal and spatial structure of the video segment. Content indexes are based on the content information of the frames that make up the video segment [42]. Video++ incorporates all three basic index types; all three index types can be mapped directly into the sections of the video description headers.

4.6 Trees and Tree Algebra

In Video++ a video segment can *include* other video segments. These video segments may themselves include other video segments. In essence, a video segment is represented as a tree whose nodes are video segments. Video++ must construct these dependency trees, referred to as *video trees*, before it can traverse them for any algebraic operations.

4.6.1 *The Tree Data Structure*

A tree is a special form of directed graph; it is a connected, acyclical graph. An **oriented** tree is one in which a vertex has been designated as the root [13]. Every vertex except the **root** has a unique predecessor. The root has no predecessors. Vertices (or nodes) that have

successors are called nonterminal vertices or parent nodes. Vertices that have no successors are called terminal vertices or **leaves**. A tree whose vertices can have an n number of successors is known as an n-ary tree.

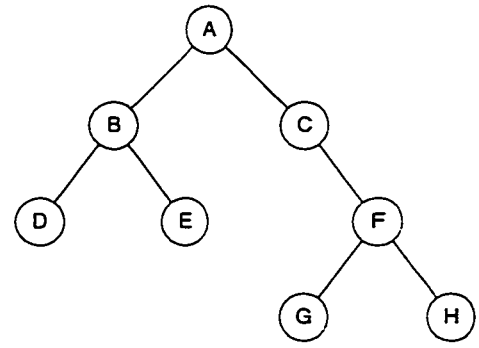


Figure 24: A tree showing terminal and nonterminal vertices

In the diagram above the root is node A. Nodes B and C have the same parent and are called **sibling nodes**. All nodes that have a parent are known as **child nodes**.

There are three basic methods for traversing trees: post-order, in-order, and pre-order [41]:

- Pre-order traversal.
  Visit the root. Traverse left subtree. Traverse right subtree. For the diagram above the sequence would be A B D E C F G H.
- In-order traversal.
  Traverse the left subtree. Visit the root. Traverse the right subtree. For the diagram above the sequence would be D B E A C G F H.
- Post-order traversal.
  Traverse the left subtree. Traverse the right subtree. Visit the root. For the diagram above the sequence would be D E B G H F C A.

4.6.2 Tree algebra

Video algebra is essentially a tree algebra. Video segments include other video segments, forming an n-ary tree. The following algebraic expressions are part of the tree algebra: addition, subtraction, union, intersection, unisection, subset, and superset.

After a video tree has been generated, it may be used in one of the algebraic operations mentioned above. The result can then be traversed and converted into a linear list. This list contains the proper temporal sequence of the video segments. The new tree should always reuse existing video segments and headers if possible.

4.6.2.1 Tree Addition

Two trees are added by simply joining their roots. The result is a concatenation of the two trees including all *duplicate* nodes.

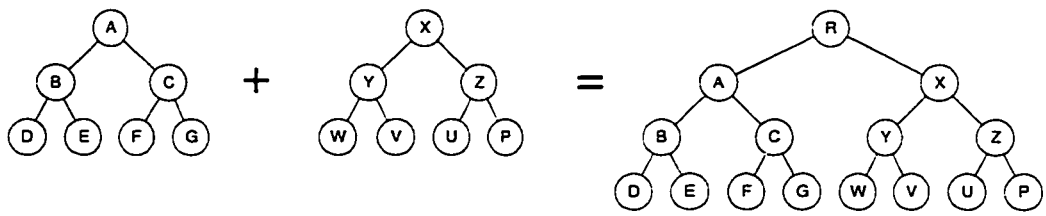


Figure 25: Tree Addition

4.6.2.2 Tree Subtraction

A tree is subtracted from another tree by removing all common nodes in the first tree. The result is the first tree minus the common nodes. In the example below, the two trees share the two nodes C and E. The result is the first tree minus those two nodes (and their children).

Some nodes have an underline in the result (such as A and B), indicating that they do not have the same children they had originally. New headers would have to be generated for these nodes if the resulting tree were to be saved.

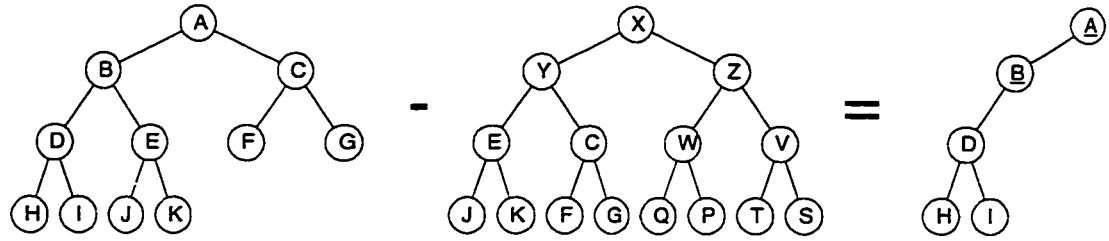


Figure 26: Tree Subtraction

4.6.2.3 Tree Union

Tree union is similar to tree addition except that duplicate nodes appear only once in the result. In the example below, the common nodes, C and E, are not duplicated in the result.

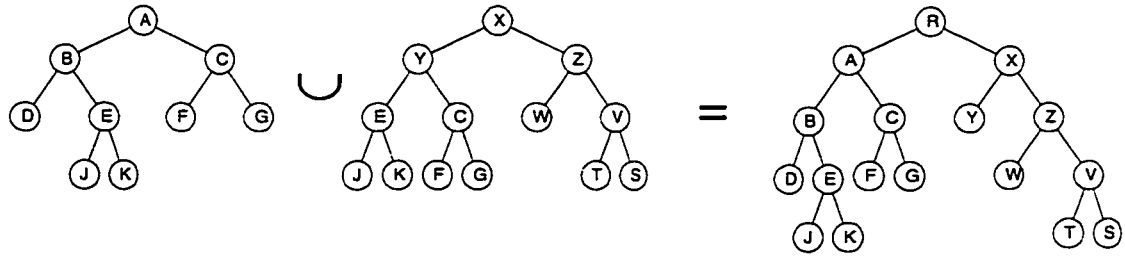


Figure 27: Tree Union

4.6.2.4 Tree Intersection

Tree intersection results in a tree that contains only the common nodes. The order of the nodes in the result is always dictated by the first tree. In the example below, the result contains only the common nodes, C and E (and their children).

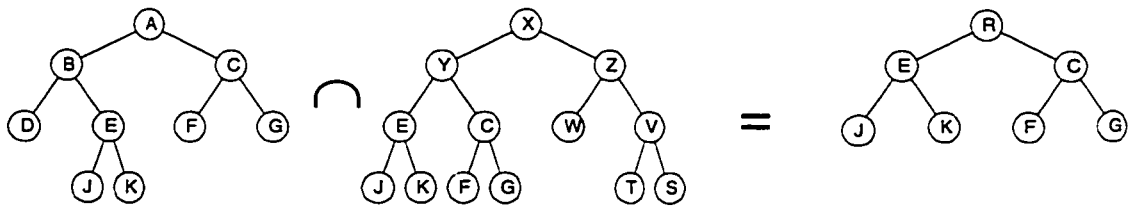


Figure 28: Tree Intersection

4.6.2.5 Tree Unisection

Tree unisection is similar to the XOR operation in boolean algebra. The result contains all nodes except those that are common to both trees. In the example below, the resulting tree has several nodes that are underlined, meaning that the tree below these nodes has been changed as a result of the algebraic operation.

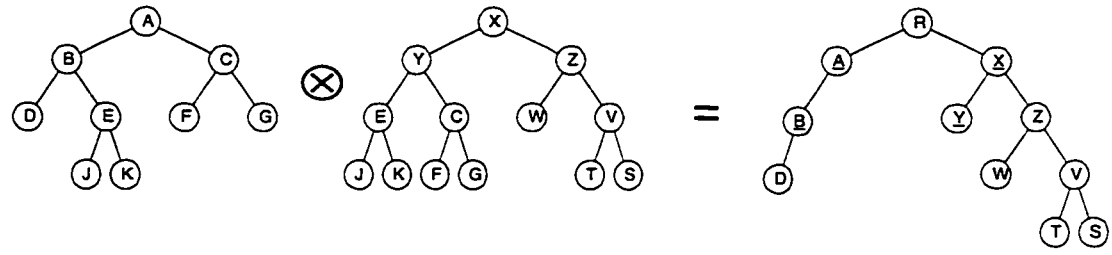


Figure 29: Tree Unisection

4.6.2.6 Tree Equality

Two trees are considered equal if they have the same nodes and if these nodes have the same order within the tree. It will be shown later that this kind of tree equality is too restrictive for video trees and that Video++ *overrides* the meaning of tree equality.

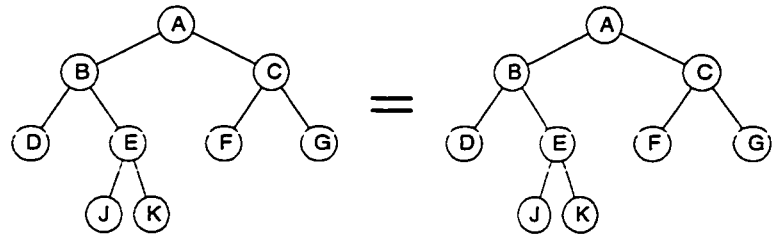


Figure 30: Tree Equality

4.6.2.7 Tree Subset

Given two trees, Tree1 and Tree2, then Tree1 is considered to be a subset of Tree2 if Tree1 is completely contained by Tree2. Again, this type of subset is considered too restrictive for video trees and has been overridden.

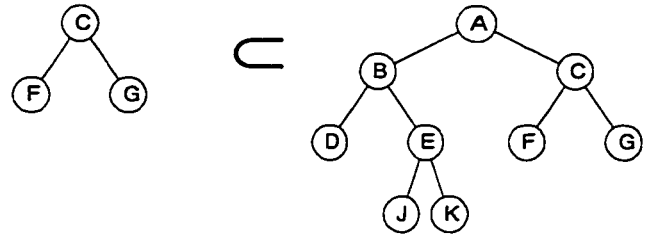


Figure 31: Tree Subset

4.6.2.8 Tree Superset

Given two trees, Tree1 and Tree2, then Tree1 is considered to be a superset of Tree2 if Tree1 completely contains Tree2. As in the equality and subset operators, the superset operator has been overridden.

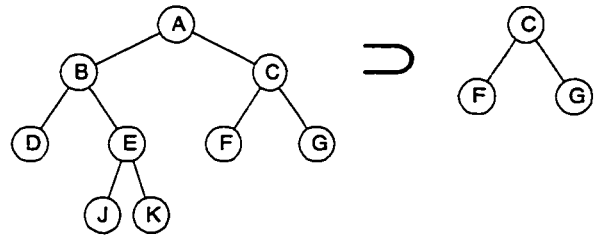


Figure 32: Tree Superset

4.6.3 The Video Tree

Video++ constructs a dependency tree for most algebraic expressions. Compound video segments and compound video headers are always parent nodes. The figure below illustrates a sample video tree which has more than one type of video description header; theses types are defined in Appendix B.

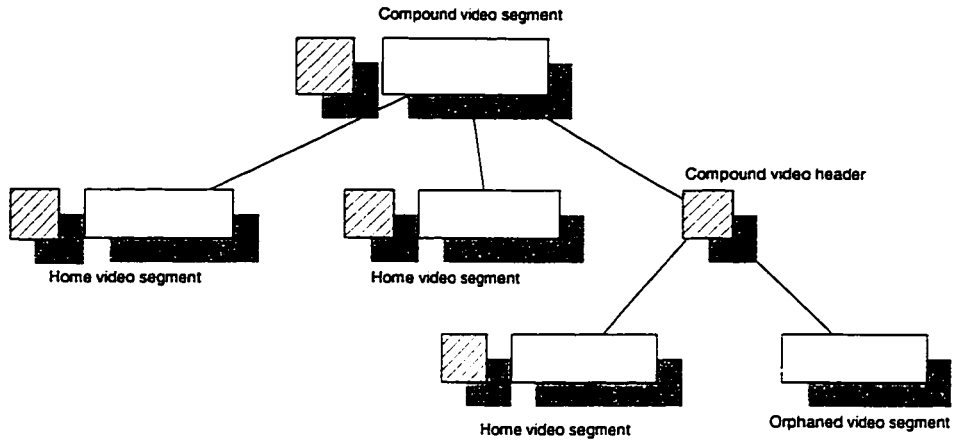


Figure 33: A video dependency tree

During playback, Video++ traverses the video dependency tree in a post-order fashion and plays each video segment in sequence. All video segments are played back in the same window. It does not matter which tree traversal method is used as long as consistency is maintained. Unless otherwise stated, Video++ uses the post-order traversal method.

4.7 Video Algebra using VML

We will now show how video description headers are used to implement video algebra. When video segments are used in algebraic operations Video++ first tries to locate the inter-video sequences. All video algebra operations are performed using the video description headers and not the actual video segments. Some of the Video++ operations may also update video description headers or generate new video description headers.

With the advent of computers, a new kind of algebra evolved. Expressions like `x=x+1` were meaningless in algebra but important for computer programming. The reason why an expression like `x=x+1` made sense is that there was a implicit temporal factor involved; the expression actually read "Add 1 to x and **then** store the result in x, the original operand". In fact, C++ provided special operators for such expressions. The expression `x=x+1` became `x+=1` where '+=' is considered as one operator. Video algebra supports similar expressions. In expressions like `video1=video1+video2` the original headers would be overwritten by

the new header resulting from the execution of the video algebra expression. The same result could have been achieved by writing `video1+=video2`.

Many, but not all, of the video algebra operations are either three-operand or two-operand operations. For example, in the expression

```
video3 = video1 + video2
```

the two video segments video1 and video2 are added and the result is stored in the new video segment video3. However, in the expression

```
video2 += video1
```

the two video segments, video1 and video2, are added and the result is stored in video2 thereby overwriting its previous contents.

The algebraic operations that have been discussed Section 4.1 will now be covered in more detail. In particular, we will explain how video description headers are used to implement these algebraic operations.

Compound video headers contain links to other video files which may themselves be compound video headers. The resulting structure is referred to as a *video tree*. Hence the operands in a video algebraic expression are video trees and the algebraic operation is a *tree algebra* operation. Home video headers and orphaned video segments can be considered to be *nodeless* video trees.

The algorithm for most video algebraic operations is as follows:

- The operands are first located on disk. If an operand can not be found, the operation is aborted.
- Each operand is then checked to see if it is a video segment or a video header.
- If it is an orphaned video segment, a single-node video tree is generated.
- If it is a home video header, a single-node video tree is generated.
- If it is a compound video header, the links are followed *recursively*, and the complete video tree for that operand is generated.

- The algebraic operation on the video tree(s) is executed using tree algebra. If it is a three operand operation, a new tree is generated; otherwise, the result is stored in the first operand by overwriting its video tree.
- Depending on the request, the video tree is sent to the display or its video tree is converted to a header and saved.

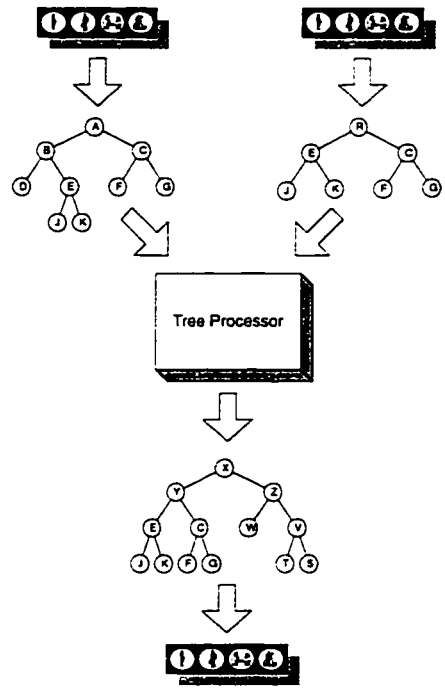


Figure 34: Video Algebra using VML

The above figure illustrates the relationship between tree algebra and video segments.

4.7.1 Addition

Header Access	Read/Write
Video Access	None

Syntax

```
video3 = video1 + video2;      // for three operand expressions
video2 += video1;              // for two operand expression
```

Process

A dependency tree for each operand, using video description headers, is constructed. The dependency trees are then added using tree addition. If the headers of video1 and video2 contain common footage (i.e. common nodes) then that footage is duplicated in the result. It does not matter if either of the operands is compressed; only the headers are affected.

The terms *shallow copy* and *deep copy* are already used in C++ and Smalltalk terminology. A shallow copy makes a copy of the original object but does not copy its parts; i.e. the parts are shared by both the original and the copy. A deep copy makes both a copy of the original and a shallow copy of the parts [28]. By default, all copy operations in Video++ are shallow copies; headers are copied but the segments they point to are never copied. In the last example we demonstrated a shallow addition. The result is not a new video segment but a new header using existing video segments. This saves on time and space.

4.7.2 Subtraction

Header Access	Read/Write
Video Access	None

Syntax

```
video3 = video1 - video2;
video2 -= video1;
```

Process

Subtraction is more complicated than addition. Any common footage (i.e. common nodes) must be identified by comparing both video trees. If there is no common footage then the result is the first operand and no further processing is necessary. If there is common footage it must be removed (subtracted) from the first tree. If the two video trees are equal, the result is a null video which is a nodeless tree.

4.7.3 *Union*

Header Access	Read/Write
Video Access	None

Syntax

```
video3 = video1 | video2;
video2 |= video1;
```

Process

Union is similar to addition except that any common footage has to be identified by comparing the video trees. Any duplicates are discarded before the video trees are added.

4.7.4 *Intersection*

Header Access	Read/Write
Video Access	None

Syntax

```
video3 = video1 & video2;
video2 &= video1;
```

Process

Both video trees are compared and only the common footage (i.e. the common nodes) is stored in the result. If there is no common footage (i.e. no common nodes), then the result is a nodeless tree.

4.7.5 *Equality*

Header Access	Read
Video Access	None

Syntax

```
if (video1 == video2)
{
   ... take action ...
}
```

Process

Video++ considers two video segments to be equal if their video trees are equal i.e. their video trees match node by node.

4.7.6 Subset

Header Access	Read
Video Access	None

Syntax

```
if (video1 < video2)
{
   ... take action ...
}
```

Process

Given two video segments, Video++ considers the first video segment to be a subset of the second video segment if the first video tree is a subset of the second video tree.

4.7.7 Superset

Header Access	Read
Video Access	None

Syntax

```
if (video1 > video2)
{
   ... take action ...
}
```

Process

Given two video segments, Video++ considers the first video segment to be a superset of the second video segment if the first video tree is a superset of the second video tree.

4.7.8 *Subscript*

Header Access	Read
Video Access	None

Syntax

```
frame1 = video1[n];
```

Process

When the video tree is generated for the video operand, each leaf at the bottom of the tree represents an independent video segment. Each of these video segments must be scanned, in order, until the n^{th} frame has been reached. The n^{th} frame may then be either sent to the display or converted to a one-frame video segment and saved.

4.7.9 *Range*

Header Access	Read
Video Access	None

Syntax

```
video2 = video1(n,m);
```

Process

This process is similar to the one in the Subscript operation. Each of the video segments at the bottom of the tree must be scanned, in order, until the n^{th} frame has been reached. The scan must then continue until the m^{th} frame has been reached. The sequence of frames between the n^{th} frame and the m^{th} frame may then be either sent to the display or converted to a video segment and saved.

4.7.10 Compression

Header Access	Read/Write
Video Access	Read/Write

Syntax

```
video1--;
```

Process

When the video tree has been constructed, each node is traversed and its corresponding video segment is compressed. If the operand is already compressed, a warning is issued.

4.7.11 Decompression

Header Access	Read/Write
Video Access	Read/Write

Syntax

```
video1++;
```

Process

When the video tree has been constructed, each node is traversed and its corresponding video segment is decompressed. If the operand is already decompressed, a warning is issued.

4.7.12 Search

Header Access	Read
Video Access	None

Syntax

```
video1("search-string");
```

Process

When the video tree has been generated, each node in the video tree that points to a video description header is scanned to search for a given search string. The search function

complements, and does not replace, existing pattern recognition algorithms. These algorithms may be invoked as Video++ callback functions.

4.8 Case Study

Before we present the actual development process of Video++, we will present a case study of one possible Video++ implementation; a typical news program. The figure below shows the structure of such a news program.

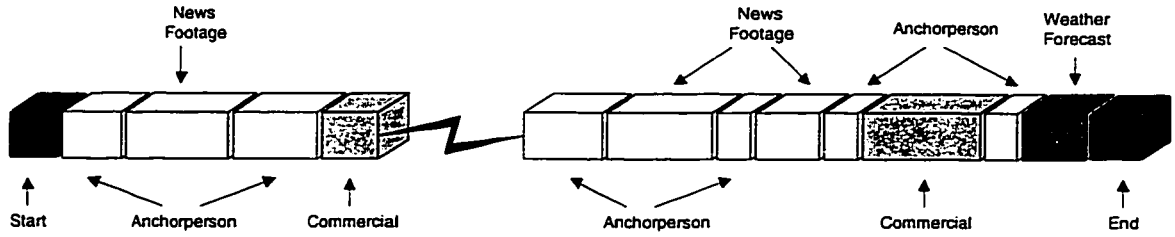


Figure 35: The structure of a news program

Individual news shots are, as a rule, not easily classified by syntactic properties with the possible exception of certain regular features, such as weather, sports, and business [19]. Parsing thus relies on classifying each camera shot according to these relatively coarse categories. This kind of syntactic analysis is the first step towards video content understanding which will enable the construction of indexes to facilitate content-based video retrieval [19].

The first step is to locate all the segments that are required for a complete news program. Each of the segments may themselves be compound segments. For example, the weather forecast may be a union of several small video segments and the result is stored in a compound header called wf1-5.10.96-10.30a.m. Let us also assume that another weather forecast is available in the compound header wf2-5.10.96-10.30a.m. Since the news programs has only one weather forecast both segments must be joined by the union operation to avoid any duplicate footage.

The following is a sample program that illustrates some of the more important concepts of video algebra:

```
#include <video++.h>

main( )
{
   // define an empty video
   video result;

   video footage1("burma_riots-23.4.96");
   video footage2("ira_bombing-23.4-96");

   video anchor1("first_segment-23.4.96");
   video anchor2("second_segment-23.4.96");

   // concatenate the two but do not allow duplication
   result = (anchor1 + footage1) | (anchor2 + footage2);

   video commercial("saturn-6");

   result += commercial;

   // display the final segment asynchronously
   display << nowait << result << endv;
}
```

4.9 Summary

The basics of Video++ were presented in this chapter. The concepts of the Video Markup Language (VML) and video description headers were introduced. The syntax of VML was discussed in detail and alternative applications for video description headers were covered. It was shown that video algebra is a form of tree algebra and a detailed introduction to tree algebra was presented. All the video algebra operations were then explained in detail.

5. THE DEVELOPMENT OF VIDEO++

The theoretical aspects of Video++ have been presented in Chapter 3; this chapter will concentrates on the implementation aspect of video algebra. An object-oriented methodology for the analysis and design of video++ has been chosen. There are currently many object-oriented methodologies available; some of the more noteworthy methodologies are GE's OMT (by James Rumbaugh), HP's Fusion (by Derek Coleman and his team), and Rational's Booch methodology (by Grady Booch). The development of Video++ is loosely based on the Booch methodology.

The development cycle of a typical project usually consists of the following stages

- Requirements and Problem Domain
- Analysis
- Design
- Implementation
- Testing
- Maintenance

Before the advent of object-oriented technology, many software cycles were driven by the "waterfall" approach which dictates that each stage has to be completed before proceeding to the next stage. To develop Video++, however, we used an object-oriented **iterative** development cycle. The development cycle is referred to as being iterative because the stages are completed in small, iterative steps; Booch refers to this method as round-trip gestalt design [5]. The figure below illustrates the difference between the waterfall method and the iterative, object-oriented method.

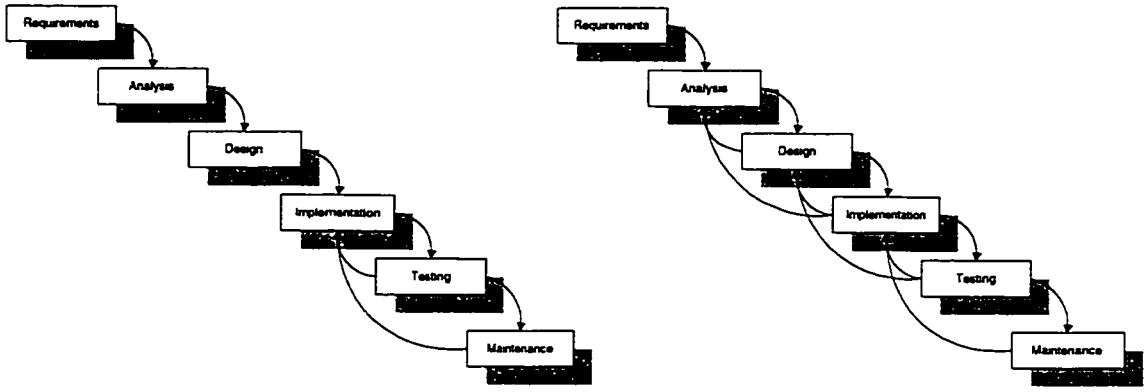


Figure 36: The waterfall method and the object-oriented, iterative method

From the above figure, the Booch method presents a set of steps that may look like a classic waterfall method. The difference is that the steps are applied iteratively. Developers analyze a little, design a little, and code a little. Then they cycle and do it again, only on more of the system. The decisions on how large each cycle must be are left to the project planners; the Booch method allows the flexibility to be as fixed or cyclic as the designers and mangers prefer [49].

The Booch method is not the only object-oriented method to advocate an iterative development cycle; the Wirfs-Brock method also iterates through requirements specification, design, implementation and testing [50]. However, there are some object-oriented methods, such as the Z++ method, that are not iterative and follow the formal waterfall method [25].

5.1 Problem Domain And Requirements Analysis

The requirements stage of the development cycle describes what the nature of problem is and the requirements to solve the problem [49]. In general, the requirements stage does not provide details of the final system. The nature of the problem has already been discussed, in detail, in the first three chapters of this thesis; video is complex and difficult to manipulate. A class library is required that will allow developers to work with video with the same ease as with intrinsic data types. In particular, the new data type may be used in algebraic and logical expressions. All the inherent complexities must be transparent to the user. It is also required to define a complete set of algebraic and logical operators that is intuitive and consistent with

the set of operators used for simple data types. The following sections discuss the requirements of Video++ in more detail.

5.1.1 Video Algebra

The first three chapters in this thesis have already discussed the algebraic and logical operations that may be applied to the new video data type; all these operation must be implemented and integrated seamlessly with the host language.

Video algebra must support at least the following operations: Addition, Subtraction, Union, Intersection, Unisection, Equality, Subset, Superset, Compression, Decompression, Subscript, Range, and Search.

5.1.2 Frames

Video algebra must be able to extract and work with individual frames. A mechanism that can convert frames to video segment and vice versa will be developed. A frame will be converted to a *single-frame* video segment while a sequence of frames will be converted to a *multiple-frame* video segment. Similarly, any frame in a video segment can be isolated and extracted.

5.1.3 Input/Output

The input and output mechanism should be kept simple and straightforward. The video output should appear in a separate window while the textual output, if any, will appear in the currently active window. As with algebraic expressions, Video++ I/O should mimic the standard I/O used for the simple data types.

5.1.4 Compression/Decompression

Two of the algebraic operations that must be supported are compression and decompression. It is also a requirement that Video++ is sufficiently open to support any of the major compression schemes. The major compression standards currently employed in the industry are MotionJPEG, H.261, MPEG 1, and MPEG 2. Other compression techniques are DVI and fractal compression [19]. Video++ allows developers to specify the compression/

decompression method they prefer. Specifying a compression technique, however, is not part of video algebra; developers will call a predefined member function in the video object to change the default compression technique. In the current release of Video++, the default compression technique is MPEG-2

5.2 Analysis

The analysis stage dictates **what** the system has to do to meet the requirements but need not show **how** the system will do it. The more obvious classes and their relationships are identified at this stage.

Ideally, the analysis stage need not know how the system is to be implemented and what programming environment will be used. The fact that Video++ will be implemented in C++ is of no relevance at this stage.

5.2.1 Video Algebra

The problem of algebraic operations involving video segments can be restated as follows:

- Assuming that an algebraic operation is to be performed on two video segments, all dependent video segments must first be identified and located.
- The algebraic operation can then be performed on the operands which usually results in a new structure of dependent video segments.
- From this new structure of video segment dependencies, one or more new video headers should be generated.

5.2.1.1 The video class

This is a public class; it's task is to act as an interface between the user and the Video++ system. One of its responsibilities is to provide an *interface* to all the algebraic operations put forth in the requirements stage. It is, however, not the Video++ class that executes these algebraic operations; that task is left for another class which is inaccessible to the user.

It must also be possible to query the state of the video objects. The video object should have member functions that indicate the different states of the video objects. The video class is one of the few classes that are accessible to the Video++ user.

5.2.1.2 The header class

It is the responsibility of the header class to generate, read, and update the video description headers. The header class does not concern itself with actually trying to locate the header; this task is the responsibility of another class, the searcher.

The header class is not visible to Video++ users. All interactions with the header class are done through visible classes such as the video class.

5.2.1.3 The segment class

The segment class contains basic information about the actual video segment stored on disk. This class would store information such as the host IP and full path name where the video segment is located, whether the video segment is compressed, and what kind of compression scheme was used.

The segment class also has the responsibility of keeping information on the file format of the raw decoded video segments. Raw video segments can be stored in one of the following ways:

- Three headerless files, one for the luminance component and two for the chrominance components.
- One headerless file with interleaved components. The component order is Cb, Y, Cr, Y. This format is also known as Abekas or CCIR Rec.656.
- TrueVision TGA format which assumes a 24-bit RGB format in one uncompressed file. Video segments in this format usually have a ".tga" extension.
- Portable PixMap format. Video segments in this format usually have the ".ppm" extension.

The figure below shows the high-level relationship between the video class and the segment class. Details of this relationship will be discussed in the design stage.

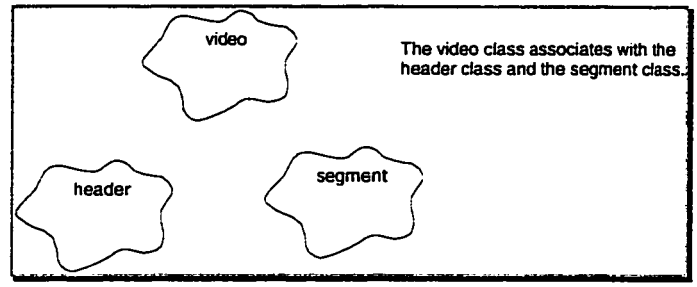


Figure 37: The video class at the analysis stage

5.2.2 Frames

The requirements state that Video++ be able to point to and extract individual frames from a video segment. The class that handles frames is the frame class. The frame class has the responsibility of holding all the information associated with a frame.

The Frame class would hold at least two important fields of information; the video segment in question and the index of a frame. An index value of 0 means that the Frame object is pointing to the first frame in the *uncompressed* video segment.

5.2.2.1 The frame class

The frame class exists primarily to point to individual frames in a video segment. The frame class also converts frames into single-frame video segment and vice versa. In the design stage it will become apparent that the frame class and the video class are tightly bound.

This class is not visible to the Video++ user. In general, frame objects are automatically converted to single-frame video objects when used in algebraic operations.

5.2.3 Input/Output

It is essential to have a proper input/output mechanism if video algebra is to have any value at all. The input/output mechanism should mimic the input/output mechanism of the simple data types. This reduces the learning curve for Video++ users.

Initially, two classes are required for input and output. These two classes are ivstream (for input) and ovstream (for output). Both these classes are public and visible to the Video++ user.

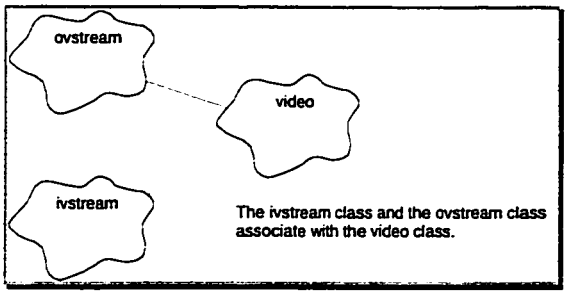


Figure 38: The vistream and vostream classes

5.2.3.1 The ovstream class

It is the responsibility of the ovstream class to accept a video segment as one of its parameters and output it to the screen in a separate window. The ovstream class must also maintain state information and provide windowing-control mechanisms.

Most C++ compilers come with the ostream class which handles most output operations for simple data types. The ostream class outputs formatted text into the currently active window. It is a requirement that the ovstream class behave as similar as possible to the ostream class. Hence, output commands for simple data types such as

```
ostream cout;    // provided in iostream.h header
int count;

cout << count << endl;
```

would have the following equivalent in video algebra:

```
ovstream display;
video toystory;

display << toystory << endv;
```

5.2.3.2 The ivstream class

The ivstream class has the task of accepting video input from various sources such as cameras, authoring tools, and even existing video segments.

C++ compilers already have a istream class which handles input for simple data types. It is a requirement that the ivstream class behave as similar as possible to the istream class. The istream class accepts input from the keyboard and echoes it into the currently active window.

5.2.4 Compression/Decompression

Compression and decompression are handled by two separate classes. The reason is that, as will be shown in the design stage, some compression techniques, notably MPEG-2, are asymmetric in nature and the compression process and decompression process are sufficiently different to warrant the use of two distinct classes. This is an example of the iterative process where decision made at later stages may reflect back on the earlier stages in the development cycle.

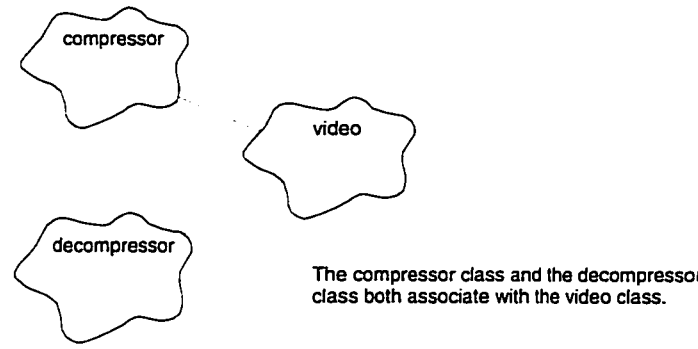


Figure 39: The compressor and decompressor classes

5.2.4.1 The compressor class

The compressor class has the task of compressing the video segment. It confers with the header class to find out if the video segment is already compressed.

The code for the MPEG compressor is available from the Internet and is written in the C language. There are two methods of integrating that code into Video++. The first method is to write an object-wrapper around the code. This is done by identifying the interfaces of the code and creating objects whose public member functions allow access to these interfaces. The second method is to reengineer the code and develop an object model from scratch. The figure below illustrates one possible (high level) class model but analysis of the code may reveal better alternatives.

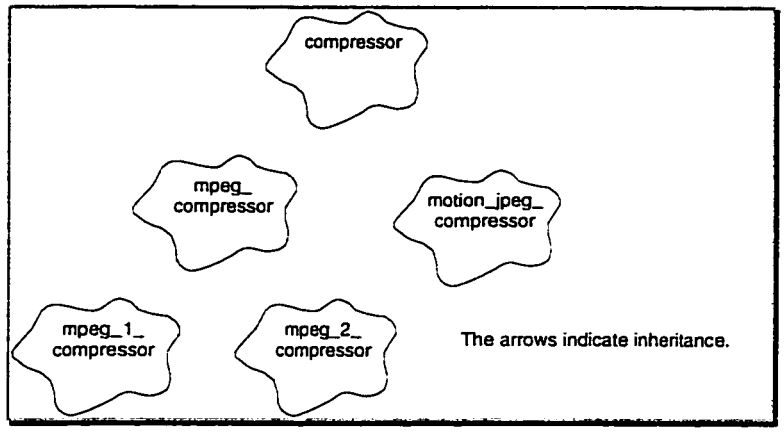


Figure 40: The compression inheritance tree

5.2.4.2 The decompressor class

The decompressor class is similar to the compressor class as viewed by other objects. Internally, however, the implementation is quite different. For example, in asymmetric compression standards such as MPEG, compression takes considerably more time than decompression and is more compute intensive. The inheritance tree, however, would be similar to the compression inheritance tree.

5.3 Design

The analysis stage dictates **what** the Video++ system has to do to meet the requirements; the design stage dictates **how** the Video++ system will be implemented.

At the design stage it already becomes important to know what programming language is to be used for the implementation. The limitations of a language can have a direct effect on some of the design decisions. The programming language of choice for Video++ is C++ because it allows the definition of user-defined types and supports the concept of operator overloading.

5.3.1 Video Algebra

In the design stage, the problem of applying algebraic operations can be stated in the following level of detail:

- Every video segment that is involved in an algebraic expression is checked to see if it is a video description header or not. For those video segments that are not headers, the corresponding home video header, if any, is located.
- For every video description header, a dependency tree is generated by reading the include section in the header files. For *orphaned* video segments, a single-node default tree is generated.
- The corresponding tree algebra operation is then applied to the dependency trees.
- If the result is sent to the display, the resulting video tree is traversed in a post-order fashion, and all terminal nodes (i.e. leaves) are displayed sequentially in a single window.
- If the result is to be saved, each nonterminal node in the resulting tree must be checked to see if a corresponding compound video header already exists. If not, a new compound video header is generated and stored in the video header pool.

5.3.1.1 Operator Overloading

C++ supports operator overloading. All languages have some kind of operator overloading but only a few languages allow the user to extend operator overloading. Java, for example, differs from C++ in not adding features to the language that make classes have the ability to exactly mimic the primitive data types [14].

In general, operators are used to provide notational convenience [46]. Expressions such as

```
F = m*a
```

are more intuitive than

```
assign(F,multiply(m,a))
```

Operator overloading allows the Video++ developer to use regular algebraic expressions as opposed to reverting to function calls. To illustrate, the addition of two video segments, using operator overloading, appears as

```
video3 = video1 + video2;
```

while, if Video++ had been written in ANSI C, the expression may have looked like

```
add_video(&video3,video1,video2);
```

The choice of algebraic operators, however, is limited. In C++ new algebraic operators cannot be created; only existing ones can be used. For example, C++ does not have an exponentiation operator and it is not possible to define such an operator [46]; one can, however, overload an existing operator. Video algebra, therefore, is limited to using operators which have a predefined meaning and with which some kind of operation is already associated. For example the plus (+) operator usually means addition. For video it could mean concatenation of two video segments. In the case of decompression, however, using an existing operator such as "++" may be not always be intuitive.

5.3.1.2 The video class

The video class is the interface to the Video++ system. As such, it was decided not include too many implementation-specific details, such as tree-related pointers, that could potentially bloat the video class. Instead it was decided to refine the class model presented at the analysis stage and add a few more classes.

When a video object is used in an algebraic operation, it is expected to deliver all the information required by the operation. In other words, it must have access to the information provided by its entire dependency tree. This information, including all tree-related functions and data has been isolated and moved to a new class, the VideoTree class. The video object maintains a containment relationship with its VideoTree object.

The video class represents an entire video tree and, therefore, cannot be used as a tree node. Another class was required to hold the information of one single node in the video tree. This class would represent either a header or a video segment in the video tree. Another class, the

VideoDesc class, was defined to act as tree nodes and to hold all the information pertaining to video segments and video headers.

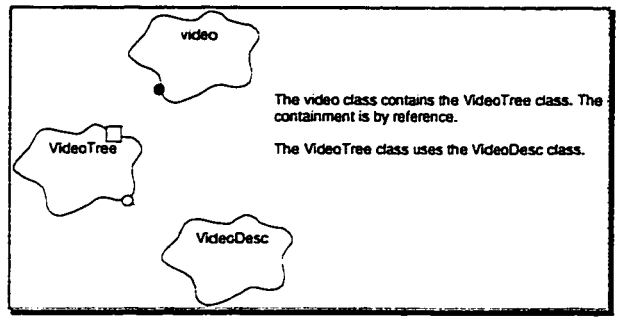


Figure 41: The video class at the design stage

Below is a fragment of the video class definition:

```
class video
{
  public:

    // constructors
    video() { }
    video(RWCString aPathName);

    // copy constructor
    video(const video&);

    // assignment operator
    video& operator=(const video&);

    // video algebra

    // equality
    operator==(const video&);

    // inequality
    operator!=(const video&);

    // addition
    friend video operator+(const video& video1, const video& video2);

    // subtraction
    friend video operator-(const video& video1, const video& video2);

    // union
    friend video operator|(const video& video1, const video& video2);

    // unisection
    friend video operator^(const video& video1, const video& video2);

    // intersection
```

```
    friend video operator&(const video& video1, const video& video2);

          ...

  private:
    RWCString PathName;
          ...
    VideoTree videotree;
};
```

Now that the video class has been properly defined, it may be used like any other data type. The example below shows a code segment

Example

```
main()
{
   int count;            // declares an integer variable
   string name;
   video graduation;     // declares a video variable
         .
         .
         .
}
```

The variables `count`, `name`, and `graduation` are all instantiations of the data types `int`, `string`, and `video`, respectively. However, of these three variables only the first one, `count`, is a true regular data type. The other two, `name` and `graduation`, are objects. For this reason, the variable `count` is sometimes referred to as an **intrinsic object** while `name` and `graduation` are referred to as **class objects** [15].

5.3.1.3 The VideoDesc class

This class was created to take off some of the work load that was expected from the video class. The VideoDesc class holds the bulk of the information on video segments and their headers. It also holds the textual information that makes up the video content and invokes the VML parser to read the video description header.

When the search operator is invoked on a video segment, the entire video tree is scanned, node by node. At each node, the search operator queries the VideoDesc object for the search string. The VideoDesc class provides the necessary search routines.

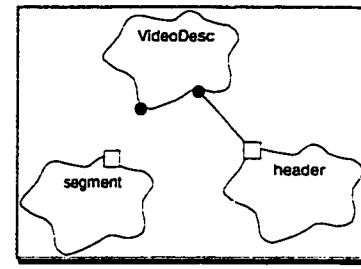


Figure 42: The VideoDesc class

Below is a fragment of the VideoDesc definition. The actual definition is much larger and can be found in the source code listings.

```
class VideoDesc
{
  public:
    // constructor
    VideoDesc();

    // copy constructor
    VideoDesc(const VideoDesc& aVideoDesc);

    // assignment operator
    VideoDesc& operator=(const VideoDesc& aVideoDesc);

    // equality operator
    int operator==(const VideoDesc& aVideoDesc) const;
    int operator!=(const VideoDesc& aVideoDesc) const;

  private:
    RWCString PathName;
    RWTPtrDlist<RWCString>   includes;
};
```

This class is not public and should not be accessible to the user.

5.3.2 The Video Tree

It is the responsibility of the tree class to construct an n-ary tree structure whose nodes represent video description headers. Since a n-ary tree is a fundamental data structure to many kinds of problems, C++ templates are used to make the tree class as general as possible. C++ templates make classes type-independent. The types are then passed to the multi-purpose classes as parameters.

In the case of Video++, the video tree consists of video nodes. The VideoDesc class is passed to the tree during compilation. Note that templates are resolved during compilation and, hence, do not cause any run-time overhead.

N-ary trees can be designed by extending the concept of linked lists. A parent would have a pointer to the first child node. All siblings would be connected via a linked list. In addition, all siblings would each have a pointer back to their parent.

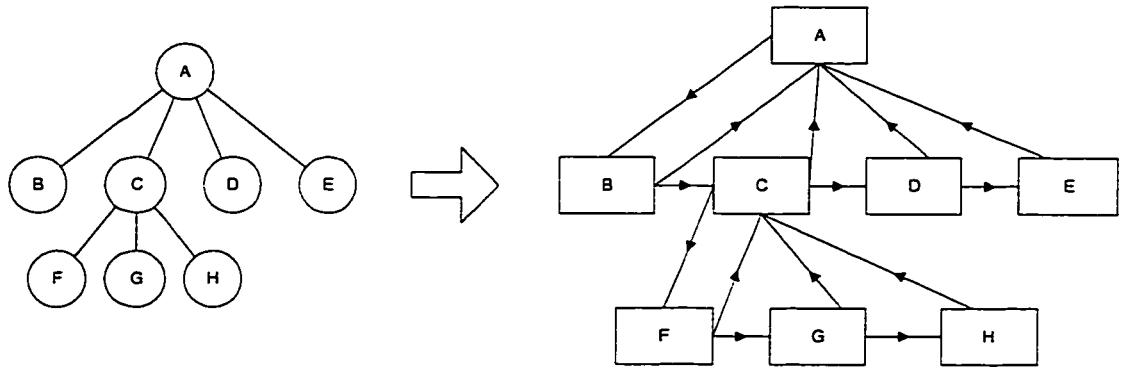


Figure 43: Tree representation using pointers

5.3.2.1 The Node class

Nodes are used by trees to hold information. The information need not be stored directly in the nodes (storage by value) but may be pointed at using pointers (storage by reference).

Below is a fragment of the Node class definition. The actual definition is much larger and can be found in the source code listings.

```
template<class T>
class Node
{
  friend class PtrTree<T>;

  public:
    // constructor
    Node(T* someinfo);
        ...
    // destructor
    ~Node();
        ...


  private:
```

```
    Node *sibling;
    Node *parent;
    Node *firstchild;
    T    *info;
};
```

The figure below shows the relationship between the PtrTree class and the Node class.

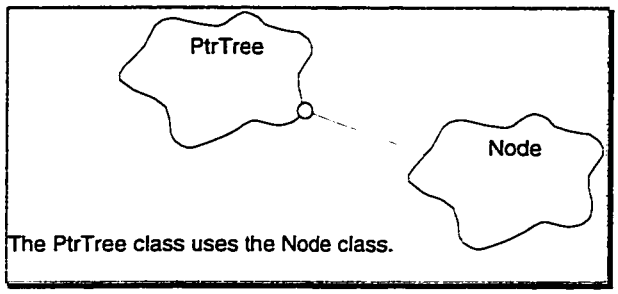


Figure 44: The Node class

5.3.2.2 The PtrTree class

The PtrTree class is a basic class that represents the tree data structure. Specifically, PtrTree must support n-ary trees i.e. trees whose nodes can have an arbitrary number of children (descendants).

Part of the PtrTree class definition is shown below. The complete definition can found in the tree++.h header file. The actual code that implements all the member functions would be found in the tree++.C source file.

```
template<class T>
class PtrTree
{
  public:

    // constants for tree traversal and tree building
    enum { PREORDER, INORDER, POSTORDER };

    // constants for list traversal and list building
    enum { INSERT, APPEND };

    // constructors
    PtrTree<T>();
    PtrTree<T>(T *someinfo);

    // copy constructor
    PtrTree(const PtrTree<T>& oldtree);

    // equality operator
```

```
        int operator==(const PtrTree<T>& sometree);

        // destructor
        ~PtrTree();

        // assignment
        PtrTree<T>& operator=(const PtrTree<T>& oldtree);

        // functions to remove and add new nodes
        void RemoveNode(Node<T>* starthere);
           ...
        Node<T>* AddFirstChild(Node<T>* newkid, Node<T>* parent=0);
           ...
        Node<T>* LeftmostLeaf(Node<T>* starthere=0) const;
        Node<T>* RightmostLeaf(Node<T>* starthere=0) const;
        Node<T>* NextLeaf(Node<T>* starthere=0) const;

        // checks if branch exhibits infinite recursion
        int isRecursive(Node<T>* starthere) const;

        // compare the relationship of two nodes
        int isAncestorOf(Node<T>* ancestor, Node<T>* descendant) const;

        Node<T>* findnode(T* someinfo, Node<T>* starthere) const;

};
```

The PtrTree class allows for easy tree traversal. It also provides the functionality of translating itself to a linear list. This is necessary when generating video headers from dependency trees.

5.3.2.3 The VideoTree class

The PtrTree class does not have the ability to construct a tree because it lacks the tree-construction algorithm. The VideoTree class, however, inherits directly from the PtrTree class and provides the additional functionality to construct trees. The VideoTree has the task of reading the header files, analyzing the include sections, and locating the corresponding video segments. The VideoTree then constructs a dependency tree using some of the member functions it has inherited from the PtrTree class.

The definition of the VideoTree class is given below:

```
class VideoTree : public PtrTree<video>
{
   public:
      // constructors
      VideoTree(): PtrTree<video>() {};
      VideoTree(video* aroot): PtrTree<video>();
         ...
```

```
    private:
       BuildVideoTree(video* aroot);
}
```

The figure below shows the inheritance relationship between the VideoTree class and the PtrTree class.

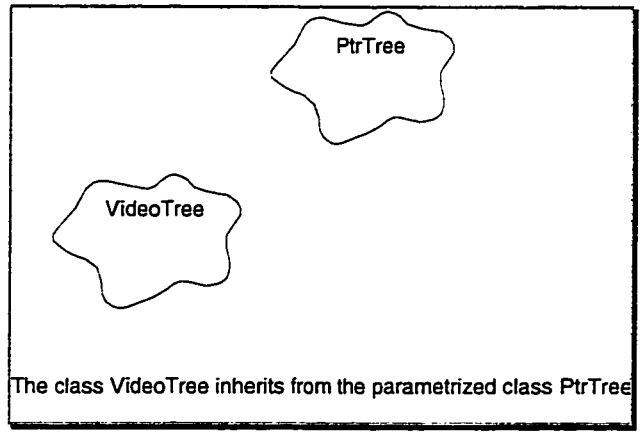


Figure 45: The VideoTree class

The figure below illustrates the conceptual model of the VideoTree class and its relationship to the Node and VideoDesc classes. The VideoTree class provides a pointer to the root of the tree. All other nodes in the tree are descendants of that root. The root itself does not have ancestors. Every node in the tree points to a VideoDesc object which contains information related to the video segments. The VideoDesc objects contain the bulk of the information; the tree itself consists mainly of pointers.

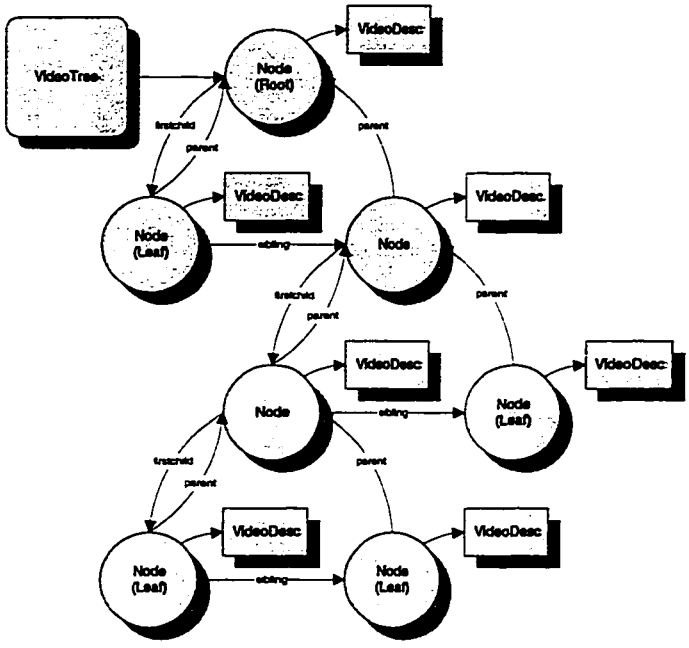


Figure 46: The conceptual video tree model

5.3.3 Object Creation and Access

An important issue that the design stage has to address is the creation of objects. Specifically, there are three functions that have an impact on how objects are created: initialization, copying, and assignment. The design stage would stipulate how the objects should behave when involved in any of these three operations; the actual implementation, however, would be left for the implementation stage.

Classes usually consist of member data and member functions. There are four member functions that have a special significance; constructors, copy constructors, assignment operators, and destructors. The last three member functions are sometimes referred to as the Big Three [12]. In general, if a class requires either a destructor, or a copy constructor, or an assignment operator, then it requires all of them [12].

5.3.3.1 Initialization

When a video object is created its constructor is automatically invoked. It is the constructor's task to properly initialize the video object. Video++ provides more than one constructor for the video class.

When a video variable is initialized with a path name Video++ attempts to locate both the header and the segment in the same directory. If the header cannot be found, Video++ assumes that the video segment does not have a header and generates a default header. If neither segment nor header are found, Video++ assumes that the path name will be later used in some algebraic expression. In other words, the path name may hold the result of an algebraic operation.

Video variables that are not passed a path name during construction are always used to hold the result of algebraic operations. Initially, the fields of the video variable remain empty to be filled later as the program executes.

Syntax

```
// initialize with pathname
video avideo("fullpath/filename");

// initialize without pathname
video avideo();
```

Examples

```
// initialize variable with header
// video++ will look for segment in the same directory
video video_syi("seven_year_itch.mv2.v++");

// initialize variable with video segment
// video++ will look for header in the same directory
video video_slih("some_like_it_hot.mv2");

// define video variable but do not initialize it yet
video video_result();
```

5.3.3.2 Copying

Copying, like initialization, deals with the construction of new video objects. The difference here is that one object is copied into another. The C++ language provides a special operator for this purpose; the copy constructor.

Syntax

```
video(const video &);
```

There are three instances when C++ automatically invokes the copy constructor:

1. When objects are passed to functions

```
bar
{
   foo(object_a);
}

foo (object_1)
{
   ...
}
```

2. When a function returns an object

```
bar
{
   object_a = foo();
}

class_a foo()
{
   return object_1;
}
```

3. When an object is initialized

```
video video1(video2)          // invoke copy constructor
video video1 = video2;        // this is not assignment
```

Note that the above statement is different from the following sequence

```
video video1;
video1 = video2          // this is assignment
```

In this case the assignment operation is invoked because video1 is not being initialized. Note that the copy constructors copies video objects and not the actual video segments.

5.3.3.3 Assignment

The developer has the ability of assigning one video object to another. In C++, assignment and initialization are considered to be different operations.

Syntax

```
video = video2;
```

Examples

```
video video1("somevideo");
video video2;

video2 = video1;          // invoke assignment operation
```

5.3.3.4 Member access

In C++, data members are usually private and can be accessed only by the class itself. If an external object requires access to this data it must go through member functions known as modifiers and selectors. These are not mandatory but it is considered sound programming practice to use them. Modifiers usually start with the prefix "set_" and allow external objects to set the values of the object's internal data. Selectors on the other hand , usually start with the prefix "get_" and allow read-only access to the object's internal data.

5.3.4 Frames

Frames are objects that are returned by some of the member functions of the video objects. Frames are only temporary structures and are never stored on disk. Frames may be considered as single-frame video segments.

The frame class has the following skeletal structure:

```
class frame
{
   public:
      int get_framenumber();

   private:
      int frame_number
      video* belongsto; // handle to the video segment
}
```

Note that the data field hold the actual image as read from the disk. In itself, this data will not be used except when it is converted to a single-frame video segment. Since frames are single-frame video segments, they can also be used in algebraic expressions.

5.3.5 Input/Output

In the Video++ system, video has a dual nature: it is represented by both, image content and text. Hence we require an input/output mechanism that handles both. The input and output of

video is complex and statements such as `display << my_video << endv;` are too simple to be useful. Video++ provides a mechanism that allows developers to control the many parameters required for video input/output.

The best way to achieve this input/output control is to use C++ manipulators [45]. The declarations for these manipulators and the declaration of the ivstream and ovstream classes can be found in the vstream.h header file.

5.3.5.1 Output

Two classes are available for the output of video segments. For compatibility purposes, the cout class will be supported but the user should always use the ovstream class when possible.

5.3.5.1.1 The cout class

The cout class is the standard output class for most C++ data types. Since it is the intention of the Video++ system to have the video type mimic the intrinsic data types, support for the cout class was added to the video class. The cout class cannot be used for the playback of a video segment but it can be used for the textual output of a video.

Expression such as

```
cout << my_video << endl;
```

produce output of the form:

```
Title = "Running from Justice"
Frame[0-200] = "Opening scene with actor John Freeman"
Frame[201-1026] = "John Freeman flees scene"
```

To have cout recognize the video class, the following member function must be added:

```
class video
{
      .
      .
   public:
      ostream& operator<<(ostream& o, video v);
      .
      .
}
```

5.3.5.1.2 The ovstream class

The requirements stage states that output should be as straightforward as that for simple data types. The analysis stage went further and states that the output should be handled by one class, the ovstream class, and all controls be handled by manipulators. The display object is instantiated from the ovstream class and is used for all output operations.

In C++, regular output is usually of the form

```
cout << data1 << data2 << ... << datan;
```

More complex output statements would include formatting information such as manipulators. It is therefore only logical that the display objects have a similar pattern. Video output expression would have the form

```
display << video1 << video 2 << ... << videon;
```

Again, the display must be controlled with formatting information such as manipulators.

The definition of the ovstream class is:

```
class ovstream
{
  public:
    ovstream();

    ovstream& operator<<(video avideo);
    friend ovstream& operator<<( ovstream&, ovstream& (*)(ovstream&) );


    enum { WAIT, NOWAIT };
    void set_waitstate(int astate);

    void set_orig(int anx, int any);

  private:
    int x,y;
    int width, height;

    // set window wait state (WAIT, NOWAIT)
    int waitstate;
};
```

The display object would then be defined as:

```
ovstream display;
```

Note that the return value of the operator<<() function is itself a ovstream class. This allows sequential expressions like:

```
display << video1 << video2 << endv;
```

Example

```
#include <iostream.h>
#include <video++.h>

video my_video("/usr/ghassan/video/pingpong.mv2.hdr");
display << my_video << endv;
```

5.3.5.1.3 Manipulators

Manipulators are a convenient, short-hand method of combining multiple I/O operation on a single line. Statements such as

```
cout << x;          // write the value of x to the output;
cout.flush();       // flush the output buffer
```

can be combined as

```
cout << x << flush;
```

In fact, any function 'foo' that has the declaration

```
ostream& foo(ostream&)
```

can be used as a manipulator for the object cout because cout is instantiated from the ostream class. The ostream class has a member function that takes a pointer as an argument. In proper C++ terminology, the type of this argument is "pointer to function taking an ostream& argument and returning an ostream&". The member function, operator<<, and its argument is shown below:

```
class ostream : public virtual ios {
   ...
public:
   ostream& operator<<(ostream&, ostream& (*)(ostream&) );
   ...
};
```

The use of manipulators, like operator overloading, is not only convenient but it also makes the code more readable.

The display object has its own set of manipulators. The ovstream class, from which the display object is instantiated, has the following member function to support manipulators:

```
class ovstream
{
        .
        .
   public:
      ovstream& operator<<(ovstream&, ovstream& (*) (ovstream&) );
        .
        .
};
```

5.3.5.1.3.1 The endv manipulator

The endv manipulator informs the display object that it should flush whatever it has in memory and shut down.

Example

```
display << my_video << endv;
```

5.3.5.1.3.2 The wait manipulator

The wait manipulator should be used when synchronous display is desired. It can only be used with the display object. It informs the display object not to return control to the next statement until the entire segment has been displayed. This is the default behavior.

Example

```
display << wait << video1 << endv;
display << nowait << video2 << endv;
```

In the above example, video2 is not displayed until video1 has run to completion.

5.3.5.1.3.3 The nowait manipulator

The nowait manipulator informs the display object to behave asynchronously and return control immediately to the next statement. This allows developers to display lengthy video segments while the program continues running.

The nowait manipulator brings with it the problem of asynchronicity; The user must perform a manual check to find out when the video segment has finished playing.

Example

```
display << my_video << nowait << endv;
   ...
while (my_video.is_playing())
{
   ...
}
```

5.3.5.1.3.4 The text_only manipulator

The text_only manipulator informs the display object that only the textual information of a video segment is to be output. This also works for single frames.

Example

```
display << my_video << text_only << endv;
```

In this example only the textual information, retrieved from the video description headers, is sent to the output. The output is directed to the current active window i.e. from which the program was run.

5.3.5.1.3.5 The video_only manipulator

The video_only manipulator informs the display object to display only the video without the textual description.

Example

```
display << my_video << video_only << endv;
```

5.3.5.1.4 Single frames

The display object also handles single frames. Expressions such as

```
display << my_video[12] << wait << endv;
```

are allowed. In this example, the frame would be shown as a still until the user closes the display window.

5.3.5.2 Input

All input is handled by the ivstream class. Unlike the display class, however, the recorder class will be hardware dependent. Different cameras will provide different APIs and that will directly affect the code of the recorder class.

5.3.5.2.1 The ivstream class

The ivstream class is similar in structure to the ovstream class. The recorder object, which is instantiated from the ivstream class, handles the input of the video segments.

The ivstream class will not be covered in more detail due to its dependence on third-party hardware. Many cameras come with their own software and sometimes provide an API that can be used from within a high-level language. If the API is not object-oriented, it is possible to add an **object-oriented wrapper** around it and incorporate it into the Video++ code.

5.3.6 Compression/Decompression

There was no need to design the compression and decompression objects. The code had been made available on the Internet by the Berkeley research group. The Berkeley MPEG-2 Encoder/Decoder Version 1.1. has been incorporated into the Video++ system with only minor modifications.

5.3.7 Error Handling

Another important issue that is usually addressed in the design stage is that of error handling. There are several alternatives of which three will be discussed here. This is just a brief introduction to error handling in C++. The reader is referred to [45] for a more detailed discussion.

5.3.7.1 Return Value

This is a vestige of the classical C programming style. All functions (and class member function) return a value that may either represent an error code or data.

```
if (myclass.foo() == ERROR)
{
   // error occurred; take action
}
```

The problem with this approach is that it is not always straight-forward to distinguish data from error codes. Another problem is that if the (called) function is deeply nested the error code will have to be propagated upward to the calling function.

5.3.7.2 Status Field

This is a common technique in C++. Most classes will have a status data member that can be written to by any member function of that class. In general, the status field will hold error codes.

Example

```
class my_class
{
   public:
      foo() { if (FALSE) status = ERROR; else status = OK; }
      get_status() { return status; }
   private:
      int status;
}

...

my_class.foo();
if (myclass.get_status() == ERROR)  // read status field
{
   // error occurred; take action
}
```

5.3.7.3 Exception Handling

This is by far the best method for capturing errors. It is also more complicated and has, therefore, been shunned by inexperienced programmers. To use proper C++ terminology, exceptions are *thrown* (a C++ term) by the function that caused the error and *caught* by the function that will *handle* the error.

Exceptions need not be propagated upward; they can be thrown by a deeply nested function and caught by the calling function. It is almost like the *non-local goto* jump found in earlier programming languages.

Example

```
class my_class
{
   public:
   foo() { if (FALSE) throw ERROR; }
}

try
{
   foo();
```

```
}
catch (ERROR) {
   // error occurred; take action
}
```

There are numerous advantages to using exception handling and Video++ makes use of it for its error handling. The main exceptions that Video++ operation throws are

- NO_VIDEO

  This exception is thrown when a particular video segment cannot be found. For example, one of the entries in the include section of a header may point to a video segment that has been previously deleted.

- NO_HEADER

  This exception is thrown when the header of a video segment cannot be found.

- NO_SHOT

  This exception is thrown when the description section of a header includes a shot which was not defined in the shots section.

- OUT_OF_RANGE

  This exception is thrown when the description section of a header references a frame or sequence of frames that do not exist.

5.3.8 The Module Diagram

A module diagram is used to show the allocation of classes and objects to modules in the physical design of the system. During development, module diagrams are used to indicated the physical layering of our architecture [5].

For the development of Video++ four modules are used. Each module consists of a body and a header. The body is usually a source file with the ".C" extension[2] while the header is a file with the ".h" extension.

---

[2] Other extensions for C++ source files are ".cpp" and ".cxx"

- The **tree++** module

  This module hold the basic classes that are required to implement n-ary trees. Specifically, it holds the definition of the Node and PtrTree classes.

- The **videotree** module

  This module uses the tree++ module to construct video trees. The VideoTree class is defined in this module.

- The **vstream** module

  This module uses the video and videotree modules to correctly input and output video trees (which represent video segments). Specifically, it defines the ivstream and ovstream classes. It also defines the input/output manipulators. This module is accessible to the user.

- The **video++** module

  This module uses videotree module. It acts as an interface between the user and the other modules mentioned above. The video class is defined in this module. This class is accessible to the user.

The diagram below shows the proper module dependencies.

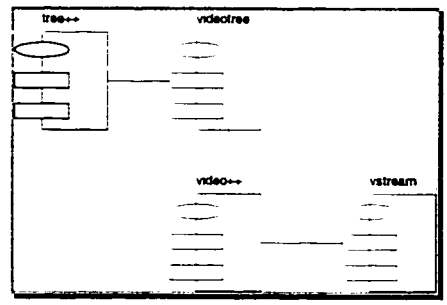


Figure 47: The video++ module diagram

5.4 Implementation

The implementation stage takes the result of the design stage and translates it into actual code. It is at this stage that developers decide on their choice of platforms, development environments, and compilers. The final part of the implementation stage is testing and debugging.

5.4.1 Platforms

In general, a good development environment would include a class browser, editor, and debugger. For the PC, some examples are Microsoft's Visual C++ and Borland C++. For UNIX workstations, some excellent C++ development environments would be DEC FUSE, Sun's SPARCworks, and HP's Softbench.

Another important factor to take into consideration is the configuration management and version control of source and object files. RCS and SCCS are standard on most UNIX workstations and provide adequate version control capabilities. More sophisticated packages include Atria's ClearCase and SQL Software's PCMS which, in addition to configuration management and version control, also provide process control.

The Video++ system was developed on a SPARCstation 20 using the Sun C++ 3.0 compiler. No version control/configuration management tool was used for the first release.

5.4.2 Class libraries

Video++ makes extensive use of general purpose data structures such as Strings, Lists, Arrays, and Trees. A class library is an object-oriented library that defines useful classes and other constructs and makes them available for other programs. These class libraries can then be linked by other programs. C++ already comes with a class library known as the Standard C++ Library . This library, for example, defines the iostream class which is used by almost all C++ programs [39]. Parts of this library are still subject to change, however, and it was decided to use another, more stable, class library. Two notable C++ class libraries were **libg++** from the Free Software Foundation (known for their GNU products) and

Rogue Wave's **tools.h++**. Both provide the source code for their class library and both provide the basic data structures required by Video++.

Video++ uses the Rogue Wave tools.h++ class library (version 6.0). This class library, even though exhaustive, does not provide any support for n-ary trees and the PtrTree class had to be written from scratch.

5.4.3 Programming

Once the tools (platforms, compilers, and libraries) have been defined the actual implementation stage can begin. The immediate input to the implementation stage is the result of the design stage. Most classes have already been defined during the design stage but the implementation stage may sometimes reveal that more classes are required.

5.4.4 The Berkeley Compressor/Decompressor

One of the responsibilities of the video class and the ovstream class is displaying a video segment in a window after or during decompression. This code, however, is available as freeware on the Internet and there was no need to reinvent the wheel. The code is written in C and runs on the X window system [19]. Ideally, Video++ would have provided an object-oriented wrapper around that code and incorporated it into the video and ovstream classes. Unfortunately, the Berkeley code was written in non-ANSI C and was rejected by the C++ compiler. The Berkeley code could only be compiled using the GNU C compiler. Therefore, the Berkeley compressor/decompressor are invoked from the Video++ system using the system() function.

The code for the Berkeley compressor/decompressor is quite large and complex. However, it is beneficial to gain an understanding of the Berkeley compressor/decompressor especially since it has been modified for Video++. Below is a brief outline (in pseudo code) of the Berkeley algorithm. The functions that are shown in the bold typeface are X Windows functions that can be found in the Xlib library. Xlib is a low-level library providing access to X graphics and interface functions. It is the programmer's main interface to the X Window System [26]. The functions, in the code below, that are not in bold typeface are user-defined functions. The first block handles the initialization, and draws the (empty) X window. It is

the while loop in the second block, however, that simulates motion but continuously calling the XPutImage() function until the end of the video segment is reached:

```
getheader()
init_decoder()
  init_display()
    XOpenDisplay()
    DefaultScreen()
    XCreateSimpleWindow()
    XMapWindow()
    XCreateImage()

while getheader()
  getpicture()
    storeframe()
      store_one()
        dither()
          display_image()
            XPutImage()          ← displays one frame at a time
  display_second_field()
    display_image()
      XPutImage()
```

The Berkeley code has been modified to work with Video++. These modifications were introduced to support manipulator control and to support displaying multiple video segments in one window. The functions most affected were `display_image()`, `XCreateSimpleWindow()`, and `main()`.

It was also assumed that the video class is sufficient in representing compressed videos no matter which compression scheme is used. This may change however, and it may become necessary to declare separate classes for each type of compressed video. For example, if a video object that points to a MPEG_2 compressed video segment differs significantly from a video object that points to a MotionJPEG video segments then the two video objects should not be instantiated from the same class. This is an example where the changes in implementation may propagate up to the analysis stage.

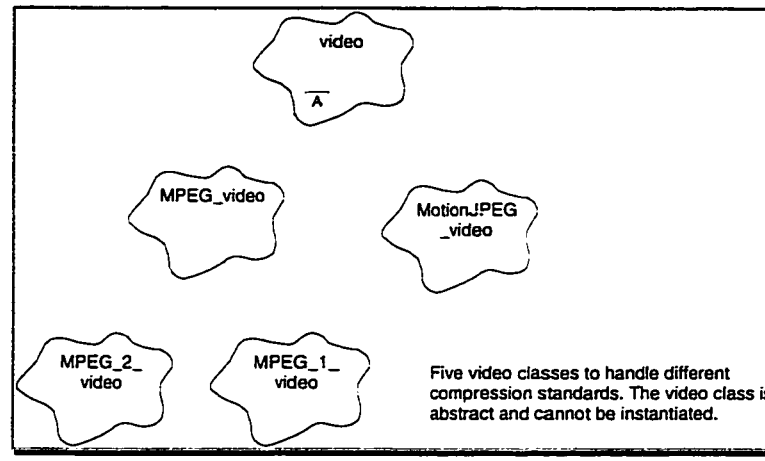


Figure 48: An alternative video class diagram

In the diagram above we have more than one video class from which objects can be instantiated depending on the compression scheme used.

5.5 Test Results

The testing stage for Video++ was performed on the same platform that was used for the implementation stage i.e. on a SPARCstation 20, single CPU configuration, with 48 MB of RAM. For the display, a 21-inch color monitor was used.

Ten short video segments were downloaded from the Internet. All were MPEG-compressed but differed in terms of height, width, and duration. Ten home video description headers were written to describe the contents of these video segments. A two level hierarchy of compound video description headers was built on top of these home headers. The links in these compound headers were changed several times during the testing phase to produce different algebraic results.

The VML syntax in the video description headers was strictly adhered to; the first release of Video++ did not have an exhaustive VML parser and would have produced unexpected results when parsing an incorrect VML statement. Future versions of Video++ should provide a more robust VML parser.

The Search operator in the first release of Video++ was implemented as a simple string comparison operator. Each line in the Description section was scanned and searched for the given keyword. If the keyword was found to be a sub-string of the Description line, the corresponding frame range (or shot range) on that same line was returned. Future versions of Video++ may use more powerful text-searching algorithms.

A Video++ test program is shown below. The program was compiled using the SUN C++ 3.0 compiler and then linked to the Video++ class library.

```
#include <iostream.h>
#include "video++.h"

int main()
{
  // define the display object
  ovstream display;

  video video_1("/video++/testpool/video1.hdr");                    ❶

  video video_2("/video++/testpool/video2.hdr");

  // find the unisection
  video uni_video = video_2 ^ video_1;                              ❷

  // display the unisection but return immediately
  // The window height should expand to accommodate
  // the full video
  display   << nowait << orig(20,30) << width(200)
            << uni_video << endv;                                   ❸

  // add the two videos and apply another unisection
  video add_video = (video_1 + video_2) | uni_video;

  display   << nowait << orig(20,30) << width(200)
            << add_video << endv;

  // find the intersection
  video inter_video =   ((video_1 ^ (uni_video - add_video))
                            + (video_2 + add_video))
                            & (video_2 | (add_video
                            ^ (video_2 + uni_video)
                            - video_1));

  display   << nowait << orig(20,30) << width(200)
            << inter_video << endv;


  video subtr_vide = (((((inter_video) | video_1) & video_2)
                        | video_1) - add_video) & uni_video;        ❹

  // display part of the intersection synchronously
  // Restrict the width of the window to only 200 pixels
  // The height of the window should expand to accommodate
  // the full height of the video segment.
```

```
    display   << wait << orig(10,10) << width(200)
              << inter_video(30,40) << endv;                    ❺

    // display the 100th frame, asynchronously
    display   << nowait << width(150) << height(200)
              << inter_video[100] << endv;


    // Search operations
     video newvideo = video_1["Driving a Bus"];                 ❻

    if (!newvideo.empty())
      cout << "The frame sequence has been found" << endl;

    display << newvideo << endv;

}
```

Line ❶ demonstrates the initialization of a video object. The header was read into memory and the entire dependency tree was constructed by scanning the hyperlinks. The entire construction process was very fast and took less than a second. Larger trees, however, would have taken longer to construct.

Line ❷ shows a relatively simple unisection of two compound video segments. The calculation of the result was instantaneous and the duration was too short to measure. This was expected because the calculation of a video expression does not require disk I/O and all the processing is done in memory. All the test programs on the SPARCstation 20 have shown that, irrespective of the complexity of video expression, the result is always calculated within a fraction of a second.

Line ❸ uses the display object to display the result of the unisection in a window. The display command uses a manipulator to restrict the window width to 200 pixels. Any pixels that extended over the right border of the window were clipped. There was no restriction on height, however, and the window height matched the height of the frames in the resulting video segment.

The display object also printed out, in text format, the result of the algebraic expression (and all the sub-expressions that were required in the computation) to the active window. This gives the user a good estimate of the size and depth of the resulting video dependency tree.

The total playback duration of a video segment can be divided into three distinct periods.

1. Opening the video file and reading the video segment into memory. This process involved extensive disk I/O and caused a noticeable delay before the video segment is actually played.
2. Initialization of the MPEG and X Window functions. This process is quite compute-intensive, and although it was not affected by the disk I/O bottleneck, it adds to the initial delay.
3. Decompression and displaying the video segment frame by frame. The duration of this process depended on the length of the video segment.

Because of the overhead in the first two time periods, the user has to wait a few seconds before seeing the first frame. There are a few methods to reduce that initial delay:

- Using a faster platform with more memory .
- Using hard disks with faster average seek times and higher data transfer rates.
- Replacing the MPEG software decompressor with a MPEG hardware decompressor.

Line ❹ demonstrates a complex video expression. The algebraic operation itself was executed almost instantaneously but the display time (including decompression) took well over two minutes. This video expression produced a very large dependency tree, with deeply, nested sub-expressions.

Line ❺ demonstrates the use of the Range operator in Video++. Only eleven frames were displayed, starting at the 10^{th} frame. The display of a range of frames may take a few seconds if the video segment is compressed; the entire video segment has to be decompressed before the range of frames is located by the Video++ system.

Line ❻ demonstrates the content-based search capability of Video++. In this particular example, we looked for the sequence of frames that depict a bus being driven. The video description header for the bus.m2v video segment describes such a sequence and is located by the Video++ search engine. After the video tree had been constructed, all its nodes were scanned for the search string. When the node was found, its DESCRIPTION section was then

parsed to locate and display the frames that were associated with the search string. The searching was very fast for the example we had chosen. Larger video trees would have taken more time to scan.

5.6 Summary

This chapter covered the entire development cycle of Video++. Starting with the requirements stage, it stated the problems and its solution in formal terms. The analysis stage then discussed the issues that Video++ has to address to meet the requirements. The design stage provided the details on how Video++ is to be developed. The implementation stage took the results of the design stage and translated it into actual code. This chapter also demonstrated that, unlike the waterfall method, the object-oriented methodology uses a more iterative approach. Each development stage may have a retroactive effect on the previous stages. The results of the testing stage showed that all the major concepts of video algebra are feasible; the implementation conformed to the theory without any major problems or workarounds.

6. COMPARISON WITH THE MIT AV SYSTEM

The MIT AV system and the Video++ system were developed independently and used different approaches to implement the application of algebraic operations to video segments. The following sections will discuss the major similarities and differences.

6.1 Similarities

Both video algebra systems, Video++ and AV, take advantage of the logical structure of video and its hierarchical relationships between video segments. This is in contrast to existing digital video abstractions, which rely on the traditional view of video as a linear, temporal medium. This section will elaborate on other similarities that can be found.

6.1.1 Hierarchical Video Structure

Both systems stress the necessity of reusing existing video segments because the sheer volume of the data makes copying prohibitive. Users should be able to create individual interpretations of existing video footage by composing new video expressions from the existing video components. Therefore, both systems address the need to have a new video data model with content-based access. The AV data model consists of hierarchical compositions of video expressions with high-level semantic descriptions. The AV data model models the complex, nested logical structure of video using video algebra. The video expressions are constructed using video algebra operations.

In the AV system, an *algebraic video node* provides a means of abstraction by which video expressions can be named, stored, and manipulated as units. It contains a single video expression that may refer to children or raw video segments.

6.1.2 Video Content

Both system use a video model that integrates both, content attributes and semantic structure of the video. In Video++, this is achieved by the use of video description headers which contain textual descriptions and also define the video structure using hyperlinks.

6.1.2.1 Text-based Descriptions

In the AV system, content is a Boolean combination of attributes that consists of a field name and a value. An example of an attribute is `title = "CNN Headline News"`. Some field names have predefined semantics - for example `title` - while other fields are user-definable. The user may add other attributes, such as actor, characters, and scene summary.

As in Video++, the AV system cannot automatically parse raw video to algebraic video nodes; currently, this has to be done manually.

6.1.2.2 Content-based Search

In the AV system, a search is performed within a collection of persistent algebraic video nodes. When the Search operation applies a query to the collection, it searches the hierarchy of every node in the collection and returns the result set of nodes that satisfy the query. Once the query result set is generated, the user can then play back any of the expressions in the set or explore the video context and composition using the operations described in the previous section. For example, the user can inspect the encompassing video segment by examining the parent nodes.

6.1.3 Video Algebraic Expressions

The AV system introduced video algebra as a means of combining and expressing temporal relations and defining the output characteristics. In the AV system, a video expression may contain composition information, descriptive information about the contents, and output characteristics that describe the playback behavior of the presentation.

As in Video++, the AV system allows for nested algebraic expression such as $((E_1 \oplus E_2) \oplus E_3) \oplus \ldots E_n)$. These algebraic operations are inherently not commutative because they include a temporal component. The same holds true for most of the algebraic operators defined in Video++.

6.1.4 Video Format

In the AV system, algebraic video files are stored in a human readable, semistructured file format and the user can edit and create algebraic video files using any available text editor.

This is similar to the video description headers used by Video++. The MIT developers are working on supporting the segmentation of raw footage using the VuSystem shot detection module and an apriori knowledge of the video stream structure.

6.1.5 Development and Implementation

Some components of the AV system were developed in C++. A set of C++ classes manages the basic functions such as synchronizing video streams, displaying in a window, and processing video streams. The only difference here is that Video++ does not process raw video streams.

6.2 Differences

The MIT AV system and the Video++ system have many similarities but, having been developed independently, they also have many differences. This section will point out some of the more significant differences.

6.2.1 Video Storage

The AV system stores and manipulates video segments that are stored in the Semantic File System (SFS) which is a storage subsystem. This is in contrast to Video++ which advocates a distributed-environment approach and *links* remote video segment together.

Furthermore, the AV system parses raw, unstructured video to algebraic video files. It does not have a concept of video description headers and, therefore, did not need to define a video markup language. Unlike Video++, the AV system cannot handle compressed video segments.

6.2.2 Algebraic Operations

The MIT Algebraic System divides the video algebra operations into four categories: Creation, Composition, Output, and Description. Video++ considers only composition and description as part of the video algebra; Output and Creation are properties inherent to the host language, C++. For example, the AV system has a special Create operator to create new

video expressions[3]. This is not required in Video++ due to its tight integration with the host language; new video expressions are created using C++ class constructors.

The AV system also has a different set of video algebraic operators. Unlike Video++, the AV system was not restricted by the C++ predefined set of operators.

6.2.3 *Video Output*

In the AV system, all video expressions are associated with some rectangular screen region in which they are displayed. A video expression constrains the spatial layout of its component. Since expressions can be nested, the spatial layout of any particular video expression is defined relative to the parent rectangle, the screen associated with the encompassing expressions. This is in contrast to the Video++ where the output characteristics are independent from the video expressions. The output operations are not considered part of the video algebra any more than the output of integers is considered part of integer algebra.

The advantage of the Video++ method is that the output characteristics can be changed without recalculating the result of the video expression.

6.2.4 *Video Content Descriptions*

In the AV system, the video model permits the association of arbitrary descriptions with a given video algebra expression. It allows textual descriptions, non-textual descriptions like key frames, icons, salient stills, and image features like color, texture, and shape. The Description operation associates content with a video expression. In the current release of Video++, only textual descriptions are allowed.

6.2.5 *Interface and Usage*

Unlike Video++, the AV system is a stand-alone application and cannot be used from within a high-level language such as C++. The AV system uses the World Wide Web and the Mosaic browser as a front-end interface. The AV system uses HTML (another SGML-based

---

[3] The MIT system is still a prototype and the Delay, Limit, Parallel-end, Transition, and Hide-Content operations have not been implemented yet.

language) to accomplish what Video++ accomplishes with VML. The Web pages serve the same purpose as video description headers in Video++. The AV system also uses the Web interface to provide special facilities for browsing video and video expressions. In Video++ this is not necessary but, if a video browser becomes a requirement in the future, it may be developed by using the Video++ class library.

6.3 Summary

This chapter provided a detailed comparison between the MIT AV system and Video++. Both systems assume that video segments can have a hierarchical structure where individual nodes in the hierarchy can be reused by other video hierarchies. Also, both systems use text-based descriptions to describe the contents of video segments. The major difference between the two systems is that the MIT system is a stand-alone application with a front-end GUI while Video++ is a class library that supports the declaration of video as a basic data type.

7. CONCLUSION & FUTURE RESEARCH

This thesis has proposed a new method of working with video using video algebra. Due to the complexities of video and its content, the concept of a video description header was introduced. The thesis has also shown that by shifting the focus from the actual video segment to its video description header, video becomes much more manageable.

To summarize, this thesis has covered the following topics:

1. It has introduced a set of algebraic operations that can be applied to video.
2. It has introduced the concept of video description headers and the video markup language.
3. It has shown how some of the more powerful features of C++, such as operator overloading, templates, and exception handling, can be used to implement a video algebra.
4. It demonstrated the complete object-oriented development cycle required to implement a video algebra, starting from the requirements stage to the final implementation stage.

7.1 Future Research

This following section will describe some of the features that may be added to Video++ in future endeavors.

7.1.1 Text retrieval

It is unlikely that automatic feature selection can act as the primary search engine. Text retrieval, on the other hand, has matured over the years and it is to our advantage to capitalize on this proven technology. One example is the World Wide Web which has proven that search by text is efficient and meets most of our demands. The searching mechanism in Video++ can be linked to existing text-retrieval software libraries.

7.1.2 *Front-end GUI*

The input mechanism in Video++ is practically non-existent. Creating video segments is a totally different technology than playing back video segments. What is required is a GUI front-end that will allow the creation and editing of video segments before they become available for use by Video++. The AV system developed at MIT [48] already provides such a front-end.

7.1.3 *Real-Time Video*

Allowing support for real-time video. The Video++ system, as it stands right now, cannot guarantee real-time performance.

Constructs such as

```
display << my_video << realtime << endv;
```

may be added in later versions of Video++ to ensure playback at more than 60 frames/second.

7.1.4 *Standard Video++ format*

Current video technology does not allow textual information to be included with the video segments. In the first release of Video++ the description headers are separate from the video segments but the headers may be incorporated directly into the video segments if the technology allows it. This would be similar to audio information which is embedded directly into the video segments.

A possible video format could be as follows:

- Every video segment will have one major header of up to 1024 characters which provides general information such as title, location, etc.
- Each sequence (a sequence is a series of related shots) will have its own minor header of up to 256 characters which describes that sequence.

The MPEG compressor/decompressor would have to be updated to take this new format into consideration.

7.1.5 MPEG compressor/decompressor

Video++ may provide its own MPEG compressor/decompressor in future releases. Reliance on third party software such as the Berkeley MPEG compressor/decompressor may cause problems in the future development of Video++. If the interface of the third-party software were to change a large part of Video++ may need to be completely rewritten. Third-party software usually comes with some form of copyright which can also prevent the wide-spread use of Video++. To site an example; in 1995 CompuServe had to relinquish its famous GIF image format due to new copyright restrictions put in place by Unisys Inc. The venerable GIF image format has now been replaced by the new PNG format but only after a major development effort by CompuServe.

7.1.6 Remote Access

The first release of Video++ supports video segments on the local host only. In future releases VML and VDH will be enhanced to include video segment on remote hosts. A possible method would be to include the IP address in the INCLUDE section. An example is shown below:

```
<INCLUDE>
   video1  marge.genie.uottawa.ca:/usr/ghassan/videos/pinpomg.mv2
</INCLUDE>
```

Another method is to deploy a new addressing scheme similar to URLs, which are used for the World Wide Web.

7.1.7 Pipelining

Currently the display object in the input/output system always directs the output to the screen. Future development must allow the output to be redirected to other applications such as video browsers, video editors, etc.

Similarly, video input in Video++ must be able to accept input from various sources.

7.1.8 Device Drivers

Video++ does not have complete control over the video segments stored on a local file system. If video segments are moved, Video++ has no way of preventing a video description header from being separated from its video segment. A device driver running in kernel mode, however, would be able to monitor all I/O operations of the operating system, recognize the operations affecting video segments, and take appropriate action.

Device drivers are usually complex and require extensive knowledge of the operating system internals. A task like this should be left to an experienced UNIX programmer.

8. APPENDICES

Appendix A: Alternative approaches to operator overloading

There are only a few languages that support user-defined language extensions. One such language is Forth [40]; another language is C++ which supports language extension via operator overloading. Video++ has taken advantage of these language-extensions but there are several other approaches to implementing a video algebra. Some viable alternatives are listed here:

- To write a preprocessor that will convert a source file to another (more standard) format. For example, a preprocessor may convert a file containing a certain video algebra syntax into a C/C++ file which may then be compiled using a regular C/C++ compiler. Preprocessors are already used in a variety of applications. A good industry example of pre-processors is Oracle's Pro*C which converts a file containing regular SQL statements into an equivalent C file (usually much larger than the original) The disadvantages of this approach is that the user does not control what the final source file will contain. This makes debugging difficult. Also, the preprocessor must ensure that the generated code is compliant with the target compiler; changes in the target compiler may result in having to rewrite the preprocessor. Preprocessors are not difficult to write and are a viable alternative for this thesis.

- To extend the existing language. A good industry example would be Object Design's ObjectStore which extends the C++ language by adding new semantics and keywords related to persistence and database management. The problems associated with extending an existing language are obvious. First, the new compiler will have to be written from scratch (no trivial task) and second, the extensions may conflict with new keywords in later versions of C++

- To write a library. Libraries are the safest way to add functionality to a language. The problem with libraries is that they usually consist of a set of routines which the user must call. This can make the code unreadable.

Appendix B: Definitions

A **home VDH** contains a description section and describes one, and only one, video segment. The corresponding video segment is known as the **home video segment.** A home VDH does not have an include section.

An **orphaned video segment** does not have a **home VDH.** During algebraic operations Video++ must generate a **default VDH** header for orphaned video segments. A default VDH are is not a home VDH and does not have a description section.

A **compound VDH** always has an include section. A compound VDH is always orphaned and can describe more than one video segment.

A video segment that shows up in the include section of a video header is referred to as an **included video segment.**

The VDH of an **independent video segment** does not have an include section. All home video segments are independent video segments.

9. REFERENCES

[1] T.G. Aguierre Smith, *If you could see what I mean...descriptions of video in an anthropologist's video notebook*, Master's Thesis, MIT, Cambridge, MA September, 1992.

[2] S. Al-Hawamdeh et al. *Nearest neighbor searching in a picture archive system*, in *International Conference on Multimedia Information Systems '91*, p.17-33, ACM, McGraw Hill, Singapore January 1991.

[3] Y. H. Ang, A. D. Narasimhalu, S. Al-Hawamdeh, *Image information retrieval systems*, in *Handbook of Pattern recognition and Computer Vision*, Chapter 4.2, p.719-739, World Scientific, Singapore 1993.

[4] Heinrich Armbruester, Klaus Wimmer, *Broadband Multimedia Applications using ATM Networks: High-Performance Computing, High-Capacity Storage, and High-Speed Communication*, IEEE Journal Selected Areas in Communications, Vol.10, No.9, pp.1382-1396, December 1992.

[5] Grady Booch , *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Santa Clara, CA: Benjamin/Cummings, 1994.

[6] Grady Booch, James Rumbaugh, *The Unified Method for Object-Oriented Development*, Version 0.8 of the draft manual.

[7] D. Bordwell, K. Thompson, *Film Art: An Introduction*, 4th Edition, McGraw Hill, New York, NY, 1993.

[8] Jeff Burger, *The Desktop Multimedia Bible*, Addison-Wesley Publishing Company, 1993.

[9] Andy Carmicheal, *Object Development Methods*, SIGS Books, Inc., 1994

[10] A.E. Cawkill, *The British Library's Picture Research Projects: Image, Word, and Retrieval* in *Advanced Imaging*, October 1993.

[11] Yee-Hsiang Chang, David Coggins, Daniel Pitt, David Skellern, Manu Thapar, Chandra Venkatraman, *An Open-System Approach to Video on Demand*, IEEE Communications Magazine, May 1994.

[12] Marshall P. Cline, Greg A. Lomow, *C++ FAQs Frequently Asked Questions*, Addison Wesley Publishing Company 1994.

[13] William J. Collins, *Data Structures: An Object-Oriented Approach*, Addison-Wesley Publishing Company, 1992.

[14] Micheal C. Dakonta, *Java™ for C/C++ Programmers*, Wiley Computer Publishing, John Wiley and Sons Inc, 1996.

[15] Stephen R. Davis, *Learn Java Now*, Microsoft Press, 1996.

[16] Janet I. Egan, Thomas J. Teixeira, *Writing a UNIX Device Driver*, John Wiley&Sons, Inc. 1988.

[17] O. Fatemi, S.Zhang, S. Panchanathan, *Optical Flow Based Model for Scene Cut Detection*, Internal Report, Visual Computing and Communications Laboratory, University of Ottawa, 1996.

[18] Alan Freedman, *The Computer Glossary*, Sixth Edition, American Management Association, 1993.

[19] Borko Furht, Stephen W. Smoliar, Hongjiang Zhang, *Video and Image Processing in Multimedia Systems*, Kluwer Academic Publishers 1995.

[20] Borko Furht, *Multimedia Systems: An Overview*, IEEE Multimedia, Vol.1, No.1, pp.47-59, Spring 1994.

[21] J.L. Gassee, *Multimedia and personal computers*, Apple Computer Multimedia Developers Conference, San Jose, California 1988.

[22] Charles Goldfarb, *HyTime: A Standard for Structured Hypermedia Interchange*, IEEE Computer, August 1991.

[23] A. Gupta, T. Weymouth, R. Jain, *Semantic queries with pictures: The VIMSYS model*, Proceedings of the 17th International Conference on Very Large Databases, pages 69-79, Barcelona, September 1991.

[24] Eric van Herwijnen, *Practical SGML*, Kluwer Academic Publishers, 1995.

[25] Andrew T.F. Hutt, *Object Analysis and Design Comparison of Methods*, Object Management Group 1994.

[26] Eric F. Johnson, Kevin Reichard, *Advanced X Window Applications Programming*, Management Information Source, Inc. 1990.

[27] Setrag Khoshafian, Razmik Abnous, *Object Orientation Second Edition*, John Wiley&Sons, Inc., 1995.

[28] Wilf Lalonde, *Discovering Smalltalk*, The Benjamin/Cummings Publishing Company, Inc. 1994.

[29] Didier Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*, Communication ACM, Vol.34, No.4, pp.46-58, April 1991

[30] Micheal Liebhold, Eric M. Hoffert, *Toward an Open Environment for Digital Video*, Communication ACM, Vol.34, No.4, pp. 103-112, April 1991.

[31] Stanley B. Lippman, *C++ Primer*, 2nd Edition, Addison-Wesley Publishing Company, 1991.

[32] W.E. Mackay, *EVA: An experimental video annotator for symbolic analysis of video data*, SIGCHI Bulletin 21, October 1989.

[33] Daniel Minoli, Robert Keinath, *Distributed Multimedia Through Broadband Communications*, Artech House, Inc., Norwood MA, 1994.

[34] A. Nagasaka, Y. Tanaka, *Automatic video indexing and full-video search for object appearances*, Proceedings of 2nd Working Conference of Visual Database Systems, pp. 119-133, 1991.

[35] Steven R. Newcomb, Neill A. Kipp, Victoria T. Newcomb, *Hytime The Hypermedia/Time-based Document Structuring Language*, Communications ACM, Vol.34, No. 11, pp. 67-83, November 1991.

[36] W. Niblack et al. The QBIC project: Querying images by content using color, texture, and shape in Symposium on Electronic Imaging Science and Technology: Storage and Retrieval for Image Video Databases, San Jose, CA, IS&T/SPIE February 1993.

[37] Uday O. Pabrai, *UNIX Internetworking*, Artech House Inc., Boston, MA, 1993.

[38] R.W. Picard, T.P. Minka, *Vision texture for annotation*, Technical Report 302, MIT Media Laboratory Perceptual Computing Group, Cambridge MA, 1994.

[39] P.J. Plauger, *The Draft Standard C++ Library*, Prentice Hall, Inc. Englewood Cliffs, New Jersey 1995.

[40] Dick Pountain, *Object-Oriented Forth Implementation of Data Structures*, Academic Press Inc., London, 1987.

[41] Editors: Anthony Ralston, Edwin D. Reilly, *Encyclopedia of Computer Science, 3rd Edition*, Van Nostrand Reinhold, New York 1993.

[42] L.A. Rowe, J.S. Boreczky, C.A. Eads, *Indexes for user access to large video databases*, Symposium on Electronic Imaging Science and Technology: Storage and Retrieval for Image Video databases II, pages 150-161, San Jose, CA, February 1994.

[43] Peter Schnorf, *Integrating Video into an Application Framework*, ACM Multimedia '93.

[44] Joan M. Smith, *SGML and Related Standards*, Ellis Horwood Limited, 1992.

[45] Bjarne Stroustrup, *The C++ Programming Language*, 2nd Edition, 1991, Reading, MA, Addison-Wesley.

[46] Bjarne Stroustrup, *The Design and Evolution of C++*, AT&T Bell Labs, Murray Hill, New Jersey, Addison Wesley 1994.

[47] Yoshinbu Tonomura, Akihito Akutsu, Yukinobu Taniguchi, Gen Suszuki, *Structured Video Computing*, IEEE Multimedia, Page 34-43, Fall 1994.

[48] Ron Weiss, Andrzej Duda, David K. Gifford, *Composition and Search with a Video Algebra*, IEEE Multimedia, Vol. 2, NO. 1, Page 12-25, Spring 1995.

[49] Iseult White, *Using the Booch Method A Rational Approach*, The Benjamin/Cummings Publishing Company, Inc. 1994.

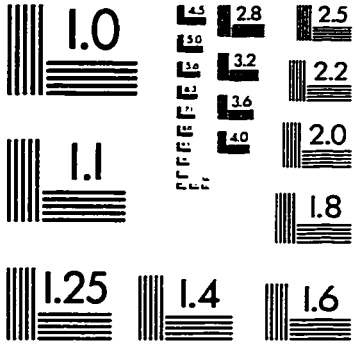
[50] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, New Jersey 1990.

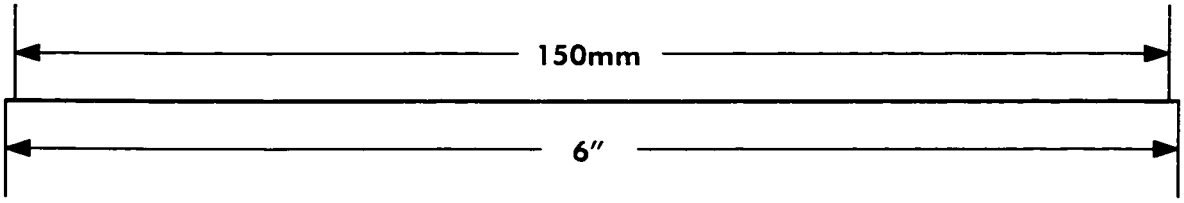
[51] HongJiang Zhang, Atreyi Kankanhalli, Stephen W. Smoliar, *Automatic partitioning of full-motion video*, Multimedia Systems, Vol.1, No.1, pp. 10-28, 1993.

[52] *IBM unleashes QBIC image-content search*, Seybold Report on Desktop Publishing, September 1991.

[53] Open High Resolution Systems Working Group, *On Harmonization of Broadcast and Non-Broadcast Uses of HRS/HDTV*, Reference document submission to CCIR Interim Working Party 11/9, 9/5/90.

TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989