A Database Representation of Motion Picture Material

by

Donovan Christopher Beauchamp

Submitted to The Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Science in Electrical Engineering

at the

Massachusetts Institute of Technology

May 1987

Signature of Author _____

Donovan Beauchamp
Department of Electrical Engineering and Computer Science

Certified by _____

Glorianna Davenport
Thesis Supervisor

Accepted By _____

Leonard Gould
Chairman, Department Committee of Undergraduate Theses

A Database Representation of Motion Picture Material

by

Donovan Christopher Beauchamp

Submitted to The Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Science in Electrical Engineering

ABSTRACT

An alternative representation of motion picture material is described within the context of an experimental video workstation. A database structure has been implemented which seeks to provide a more intuitive level of association between visual information and thematic content. As electronic archives become a common resource, the tools to provide a variety of viewing experiences with the same video materials must also evolve. The proposed workstation environment seeks to address this growing need.

Thesis Supervisor: Glorianna Davenport

Title: Lecturer of Cinema

Contents

1 Introduction **4**

2 Background **7**
2.1 History of the Technology 7
2.2 Videodiscs 8
2.3 Relegating Control to the Computer 9

3 Database Representation **12**
3.1 Hardware 12
3.2 Database Model 14
3.2.1 Seamless Transitions and Preview 17
3.3 User Interface 19
3.3.1 Keyboard Commands 19
3.3.2 Graphical Interface 20
3.4 Links 20

4 Future Work **22**
4.1 Graphical Interface 22
4.2 Levels of Hierarchy 22
4.3 Transitions 23
4.4 Preview 23
4.5 Videodisc Agenda 24
4.6 Database 25

5 Conclusions **27**

A Ingres Database Summary **29**
A.1 Database Creation 29
A.2 Tables 30
A.3 Loading Data 31
A.5 Queries 32

B GDB Summary **34**
B.1 Building the Bridge 35
B.2 Bobcat Implementation 39

C Figures **41**

D C Programs **45**

References **75**

Chapter 1 Introduction 4

The problem of efficient access to motion picture materials is one that has traditionally been the concern of film industry and television broadcast editors. This is primarily due to the prohibitive expense of comprehensive editing systems as well as the need for detailed knowledge of video technology. With the advent of micro and mini-computer technology as well as the introduction of playback devices for the novice videophile, the ability to create many different viewing experiences from a single set of video materials is now possible in a less cumbersome environment. In order to explore what attributes such an environment might possess, certain characteristics of the medium require review.

Many film historians draw a distinction between two primary genres, the narrative film and the documentary. While the narrative film is generally characterized as the weaving of a tale, the documentary is more appropriately described as the cataloguing or observation of an actual event or process. In some sense, the unifying thread between the two genres is the artistic expression or embellishment provided by the directorial and editing team. However, one area in which they differ is in their preferred mode of presentation. Specifically, the narrative film assumes a *linear* presentation. This is quite distinctive from the *interactive narrative movie*, where the viewer chooses between several pre-defined plot paths at different points in the film. In narrative film, there is no arbitrage situation; the viewer's role is

a passive one where all the decisions as to the order of presentation have been made in advance. The issue of presentation format is particularly sensitive in the area of documentary film. This is due to the unfortunate tendency to relegate significant amounts of film footage acquired during a shoot to inaccessible archives or simply to allow them to be erased because they were not immediately utilized in the initial presentation of the material. This is partially attributable to the nature of the medium itself. In order to provide many versions of a film, whether documentary or narrative, the responsibility has traditionally fallen on the filmmaker to re-edit the footage and to record the new version onto a different reel of film or videotape. Since this editing process is always lengthy and often tedious, few filmmakers produce more than one version of a given film. One solution to this problem would require that the viewer have a means to reconfigure the film that does not depend upon a familiarity with traditional editing equipment. In this way, the raw footage could be viewed in its entirety and formatted according to the needs of a particular viewing audience as many times as desired. Professor Richard Leacock aptly refers to the documentary as a *fishing expedition*; the workstation environment herein described seeks to provide the viewer with a parallel experience. Thus, while not advocating a purely interactive documentary style, the current linear form of presentation seems to be too limited for both filmmaker and viewer.

Altering the configuration of a film implies a means of representing contiguous pieces of that footage. In this way, any film presentation could be viewed as one of any number of ways to order these smaller blocks. The traditional approach has been to use shot lists and a numerical indexing scheme embedded in the medium itself. However, the reliance upon a numerical method often separates the filmmaker or viewer from the content

and context of the footage. Excessive time is spent in linearly reviewing the material as well as maintaining hand-written logs of shot placement. A method of allowing one to associate ideas and perceptions of the footage as well as some form of visual representation of the film would certainly be more satisfying. It would also provide a more intuitive way of connecting smaller parts of the film together.

To this end, an experimental video workstation has been developed which seeks to provide an alternative means for both filmmaker and viewer to study film content and to edit large amounts of video materials into logical, expressive presentations. One of the primary charters of this system has been to create an environment that alters the perception of motion picture materials from a fixed, inflexible arrangement of images to a visual resource capable of being restructured according to the intent of the viewer or filmmaker. It is hoped that this would *personalize* the viewing experience, providing different audiences with unique delivery formats. This paper details a representation of video that provides both an efficient means of access as well as a level of conceptual association. Section two reviews the history of technological advancement that has made the implementaion of the workstation possible. Section three outlines the database representation as well as the constituent components of the system. Section four suggests areas for future work and possible strategies.

Chapter 2 Background 7

In order to establish a relationship between the film medium and a database representation via computer, it is useful to review some of the key points in the evolution of video technology.

2.1 History of the Technology

The Ampex Corporation developed the first Video Tape Recorder[1] in 1956. The first application of this device was the recording of programs in New York for later playback on the West Coast. Prior to the advent of VTRs, the method of *kinescoping* was used; this was the technically inferior process of filming the programs directly from a TV monitor. Editing, at first, consisted of splicing, but soon the advent of electronic editing with a record and playback VTR was the sole approach. Previewing[2] edits was not possible with early systems.

One of the major developments of the 1960s was the introduction of *time code* in 1967 by EECO, a large manufacturer of professional video editing equipment. This allowed every inch of videotape to be indexed by a time code expressed in hours, minutes, seconds and frames. Due to the increasing use of time code and an excessive number of emerging formats, in

---

[1]The first VTRs were of 2 inch format; since then, many other formats have appeared. Among them are 1-inch, 3/4 inch and 1/2 inch formats, some of them open reel and others cassette

[2]The method of previewing edits refers to the process of deciding what the order of shots in a given scene will be and viewing the sequence without actually recording the sequence to tape.

1969 SMPTE(The Society of Motion Picture and Television Engineers) introduced an industry standard for timecode that was adopted in the United States and by the EBU(European Broadcasting Union). The National Television Standards Committee(NTSC) time code reads from 00:00:00:00 to 23:59:59:29, recycling at each 24 hour interval. The time code is recorded as a binary signal and designed to generate a complete line of information twice during each frame. Each category(hrs, mins, secs, frames) is comprised of two 4-bit words; one word in units and the other in tens. Thus, all possible values may be represented. Longitudinal Time Code(LTC), recorded on the audio track of videotape, is normally 80 bits in length. Among these bits are usually 8 four bit words that are "user" bits and may be used to indicate anything desired by the production team (reel number for example). When doing a shoot, the time code generator and the video signal must be in sync. There are 16 bits of sync information on the time code to do this. There is also a drop-frame bit which is used to compensate for the fact that the NTSC signal operates at 59.94 Hz. and the clock normally runs at 60 Hz. The difference amounts to 108 frames per hour. Normally the fix entails skipping two frames a minute except for every tenth minute. If using non-drop frame mode the rule of thumb is that 30 minutes of programming is actually 2 seconds longer.

2.2 Videodiscs

One of the most significant events in the 1970s was the release of optical videodisc playback devices. Video information on a videodisc is stored in very small pits pressed into spiral tracks on a reflective surface. Two layers of clear acrylic plastic, on either side of the disc surface, serve as a protective

barrier. Although these discs resemble phonograph records, the method by which the information is read is quite different. As the disc revolves, a laser beam reads the information encoded in the pits of the reflective surface. The beam is reflected off the surface and sent to a mirror which in turn is reflected to a microprocessor decoder which converts the information into image and sound. Since the beam acts as a non-contacting *stylus*, there is virtually no wear on the disc or image degradation over the course of the disc's lifetime.

The CAV(Constant Angular Velocity) disc format is the most widely used in editing system applications due to the fact that each frame of video is addressable. These discs spin at 1800 rpm reading 30 frames per second, meeting the NTSC standard for video. Up to 54000 frames may be stored on a single side of a CAV disc. This may be used to store 54000 still images or 30 minutes of continuous play video at 30 fps. Embedded in each frame on the disc is an index, or frame number. This allows the access capability afforded by SMPTE time code but at much higher access speeds. The current maximum delay in moving from one point of a disc to another is approximately 2.5 seconds. One of the drawbacks of current disc technology is its read-only characteristic. This is the primary reason that VCRs have been more successful in the consumer market. However, the accessing modes,durability, and memory capacity(3 Gigabytes) make the optical disc player a much more useful tool in the educational environment.

2.3 Relegating Control to the Computer

In the context of the technological advancement summarized above, two issues become apparent. The first is that there is a lack of order within the

traditional methodology for processing motion picture material. One possible reason for this is a lack of standardization between devices that perform the same function. One might attribute this to the absence of a well-governed organization that also has the influence to define standards to which all developers and manufacturers in the industry would adhere. However, this view is only partly true since it does not address the complexity of the video signal itself. The empirical techniques employed to maintain and enhance the quality and integrity of the video signal has created industry professionals that are one part engineer, one part artist, and one part magician.

The second issue is related to the emergence of timecode in the industry. This numeric index of the contents of a piece of video has opened a door to a more orderly approach. And yet those who create films have only partially stepped through this passageway. The virtue of time code is that it allows the incorporation of a computer in two arenas. The first is that of controlling the devices used to manipulate video, while the second encompasses the potential for a parallel representation of the material. An unbalanced amount of effort has been spent upon device control and the bookkeeping involved in the management of many generations of editing lists, while work in the latter area has been slow and isolated. One of the goals of the workstation herein described is to provide an environment that supports the required device control as well as a foundation from which representational constructs of motion picture material may be developed. In the next chapter, a system having a database representation of video as one of its principal components is described. It is our hope that, aside from

relegating numerical indexing to the computer, greater modes of association, both textually and graphically, will be possible.

Chapter 3 Database Representation 12

The workstation described in this chapter has had many contributors. Much of the conceptual groundwork, upon which this implementation is based, has evolved gradually over the last several years. As more hardware found its way into the lab, different approaches on a variety of computers were employed to address the issues of device control and interfaces. While the current system is in some sense a culmination of much of this previous work, the following summary is at best a snapshot of a dynamic process that continues to expand in scope and in depth. The workstation consists of three primary components:

1 - Hardware
2 - Database Model
3 - User-interface

Each of these areas will be discussed separately, followed by a description of the way in which they link to one another.

3.1 Hardware

The workstation consists of the following devices:

HP(9827) Bobcat computer : This computer has a UNIX operating system. All system software has been written in the C programming language. There are four external ports for the integration of video devices as well as a separate graphics monitor.

Sony CAV Optical Videodisc Player: Two of these players are connected to the Bobcat computer via the RS232 serial interface.

CMX Audio-Follow-Video Switcher: This device switches between the two videodisc players and provides various ways in which to perform an aesthetically pleasing transition.

Sony Trinitron Video Monitor: This is where all video is displayed.

IBM PC/AT: A frame buffer has been installed into the PC which provides the capability of digitizing a video image and performing various manipulations on the stored image. These images will be shipped via ethernet to the Bobcat and displayed on the Bobcat graphics screen.

The configuration for the System is presented Fig. 1

In a previous demonstration at the National Association of Broadcasters conference last spring, a video multi-viewer was also proposed as a useful addition to the hardware configuration. It is felt at this time that this device is not necessary in achieving the goals of the system. There are also several *special effect* devices currently available in the laboratory. However, the time it would take to completely understand their functionality

and the way in which they might be controlled by the computer (writing a driver for each of the devices) has led us to postpone their incorporation for the time being. The absence of a record video tape device should also be addressed.

The emphasis at present lies in the viewing and representation of the motion picture material rather than on the recording of a specific session. Of course, the integration of a record VTR has been anticipated in the design of the current system in such a way that future incorporation should not pose a major problem. One of the major design goals was the creation of an open system that would allow for the integration of additional devices with ease. One example of this is the number of disc players being used by the system. At present we are using two; theoretically, the system is designed for **n** disc players. The only real limitation is the number of ports available on the Bobcat. Further, the video switcher employed at present only supports cuts. Once a switcher that supports the more elaborate types of transitions(wipes, dissolves) is acquired, a driver will have to be written before it can become part of the system.

3.2 Database Model

It is in representing film material as storable and retrievable data that the ability to reconfigure the footage an infinite number of ways emerges. The configuration is made possible by the presence of some form of time code embedded in the medium. In the case of CAV videodisc, the index is the frame number. Thus, a linear piece of video on an optical videodisc is defined by the frame number at which the shot begins and the frame number at which it ends. If these in/out points are stored in some permanent memory

location on a computer, the question of how to associate other types of information with them arises. This issue of association was addressed by the introduction of a relational database into the system architecture.(See Fig. 2)

A relational database provides the means of partitioning large amounts of data into smaller, more coherent pieces. Of the many databases commercially available, the RTI Ingres Relational Database package was chosen. The decision to use Ingres was based upon its superiority over other available databases as well as the fact that it is generally available and supported by MIT's Project Athena.[1] The wide variety of applications provided by Ingres necessarily imply a relatively complex interface to its functions. However, the primary goal of the workstation was to provide the simplest, most direct database representation of video materials. Therefore, only the most basic and straightforward of Ingres' capabilities were required.(See Appendix A) The *Database Manager* in the figure refers to a buffer between the user and the Ingres command interpreter. It provides the user with the most useful of the features supported by Ingres, while at the same time hiding the syntax from him. The information associated with these in/out points will be of two primary types:

1- Textual information: This can include any textual information that the user feels is relevant to a piece of footage. Some examples might be: transcripts of dialogue, names of characters, a thematic category or perhaps simply a short list of comments.

[1]Project Athena is a five year development strategy to explore computing environments in an academic setting. It is funded by a variety of private sector organizations.

2- Graphical Information: This refers to a digitized image captured in this system by the IBM PC. The Targa 16 board allows a user to compress an image into an icon which be displayed on the HP Bobcat graphics screen. These icons are intended to serve the purpose of a *memory-jogger* for someone who has already viewed the video material rather than as a conveyor of new information.

The atomic unit inside the system will be referred to as a segment. Within the database, a segment refers to a unique record that contains the in/out points of a piece of video as well as any textual or graphical information associated with it. As a first step in exploring what types of information would be generally useful to both filmmaker and viewer, a database entitled *videofile* has been created with the following categories(See Appendix A):

1. Disc Name
2. Segment Name
3. In Point
4. Out Point
5. Theme
6. Comments
7. Character List
8. Date
9. Icon Address

Once a set of video segments has been represented and stored in this fashion, the different modes by which one may access and manipulate these

segments increases dramatically. The major reason for this lies in the utilities provided by the Ingres database system. The ability to perform a query is perhaps the most powerful of these utilities. A query allows the individual to search through an arbitrarily large amount of information stored in a database and view only those records(segments in this application) that satisfy a user-defined criteria. In this way certain patterns may emerge that otherwise might go undetected. An examples of this might be the querying of the database according to some thematic category and seeing whether the characters in a scene or the date in which it was shot have any correlation.

Another significant benefit from this representation is the ability to put together a sequence of segments without having to deal with the numerical indices(frame numbers) that define them. If we assume that a disc name and a segment name uniquely define a particular segment in the database, then by knowing those two pieces of information, the system can search the database and find the in/out points associated with that particular segment. By the same token, if we have an visual icon associated with a database segment, selecting that icon will allow us to get at the frame numbers, or other relevant information such as character names, dates etc.

3.2.1 Seamless Transitions and Preview

Once a sequence of segments has been defined, the system will translate the sequence into an Edit-Decision-List format. The most prevalent format of editing list in the film and video industries is one developed by the CMX corporation.(See Figure 3) This format associates time code with a particular type of transition between shots. Each edit within a CMX list is referred to as

an *event*. The analogy to the popular CMX EDL format will be tranparent to the user unless he wishes to actually view his defined sequence in this format. These EDLs define the viewing experience. Originally, we contemplated having a facility that generated an EDL on the fly as the viewer scanned through each segment. We no longer feel that this option is necessary or even desirable since the viewer will have the option to configure the segments in any way he chooses.

The issue of whether to have a local database structure identical to the larger system database has also arisen. The idea was to allow the user to preview edits without saving them to permanent memory. If the specified sequence of segments was acceptable, it could then be saved in the permanent database by the user. We have decided to reverse this strategy so that all segment definitions are in fact written to the permanent database, while the list of segments that define a sequence is saved to a text file when created. This approach seemed more appropriate since segment definition and sequence definition are two logically distinct processes.

We wish to address the issue of seamless editing transitions here. It was felt that using two playback devices with identical videodiscs would allow us two perform a seamless, smooth transition between video segments. The algorithm developed to preview a user-defined edit list has been successful in this respect. A seamless transition is achieved by cueing the first disc player to the in point of the first segment and then playing the segment. While the first disc player is playing, the second disc player will cue to the in point of the second segment . Once the first disc player has reached the out point, the input to the monitor is *switched* to the second disc player and the second disc player is now instructed to play. This is repeated until the end of the list is reached. In this way the facility of editing motion picture material stored on

videodisc is achieved.without having to focus on the numerical indices. This algorithm hinges upon the ability for a disc player to cue to a particular point on a disc almost instantaneously. This is possible since the disc players are spinning at speed even when they are not playing.[1]

3.3 User Interface

The user interface for the EVW consists of two parts:

1 - Keyboard Commands
2 - Graphical Interface

3.3.1 Keyboard Commands

Almost all computer-controlled video editing bays have all the available editing options assigned to the standard character set on the keyboard. A representative keyboard is that of the Grass Valley Group System. One of the ways in which a user interfaces to the EVW is with keyboard commands very similar to the assignment of the GVG system. In some sense, this mode of control is the most fundamental since it has the virtue of being portable to any other keyboard and thus would be the manner in which a remote user would access the system. However, until an effective strategy has been defined to allow a remote user to view the video material (Cable TV channel assignment, for example), the user will necessarily have to use the

[1]This differs from traditional CMX tape previewing options. When previewing a sequence of shots with tape-editing equipment, it is necessary to *pre-roll* the players to the edit point; this allows them to get up to speed before the edit is executed.

laboratory version of the system. The standard set of keyboard options will include: forward play, cueing and rewinding at various speeds, and segment definition.

3.3.2 Graphical Interface

Once it has been established that basic functionality will be possible with a standard keyboard, we would be remiss if we did not utilize the additional features available on the HP Bobcat to their fullest potential. Among those features are a mouse interface, a keypad, a knob panel, and a tablet. The Bobcat monitor also displays very impressive, crisp color images. With these features and the use of a windowing package, a complete graphical interface may be developed allowing the user to perform all those functions available via the keyboard with mimimal use of the keyboard. Computer graphics will be used to visually represent the physical processes of editing the video material and controlling the video playback devices. As stated previously, digitized key frames will be generated through the Targa board on the IBM AT. The only planned constraint is that the user be provided with the ability to generate a keyframe only for the purpose of associating it with an already defined video segment. A pointer to the location of the key frame in memory will be stored in the system database.

3.4 Links

Now that the separate components of the system have been described, we may turn to the way in which they interact with each other. The first step was assigning keyboard command for each of the viewing options supported by

the disc players. This is done within a C program by sending the command to the device across an RS232 serial port.[1] The options provided by the database manager have also been provided by keyboard assignments. This was also accomplished within an application program using a set of functions (GDB) that allowed communication between the Bobcat computer and a remote computer on the network where the database was created.(See Appendix C) Therefore, the *first level link* between these two systems is the keyboard. The ability to preview a sequence of database segments is composed of two parts. The first entails a means to load in/out points from the database into a text file in an EDL format. The second translates this list of in/out points into a set of device commands(cue, play, switch etc.) that are sent to the appropriate disc players. The next logical step in linking these routines is through a graphical icon-based user interface. This could be built on top of the existing keyboard architecture to allow input to come from a mouse or other input device rather than directly from the keyboard. Work in this area has already begun and is viewed as necessary since it provides a visual interface to a visual medium.

[1]The program to assign all disc player options to characters on the keyboard was written by Reza Jalili '89.

Chapter 4 Future Work 22

4.1 Graphical Interface

The system described in the previous chapter , while providing the basic functionality desired, will benefit greatly from additional work in several areas. The most important area is perhaps that of the graphical user interface. The problem of creating icons on the IBM AT is now well understood. However, work in the area of displaying these icons and creating well-organized windows of these icons has only reached a preliminary phase. The issue of providing hooks between these icons and records in the video database is also an important area where work should continue. Currently, there is a field in the database that is entitled *icon_address* . The thought was that this field could be used to store a location in computer memory where the actual icon would live. An alternative might be the use of a unique icon name that could be stored in the database record instead. Some type of icon table would then need to be created to associated a region of the graphics screen with an icon name so that selection of the icon would be immediately translated to a selection of a single record within the database.

4.2 Levels of Hierarchy

One area that has purposely been ignored for the time being is that of creating a hierarchical structure for sequences of video segments. Currently,

once a sequence has been created using the preview capability, there is no way in which to use this object as a component of another sequence. It was felt that this hierarchy would best be represented pictorially with the graphical user interface. A different type of icon might be used to distinguish between different objects in the hierarchy. Russ Sasnett has done some very interesting work in this area using a film strip and index card metaphor to distinguish between different classes of objects.[1]

4.3 Transitions

The different types of transitions that one might want to use in order to create a video presentation are inevitably limited by available hardware. The recent acquisition of a new video switcher will make it possible to increase the number of different types of transitions as soon as a device driver is written. Among these will be wipes, dissolves, and fades

4.4 Preview

In the area of previewing a sequence of pre-defined shots, some additional routines would be beneficial. Currently, no checking is done to avoid playing redundant in/out points. For example, if a sequence of two segments is to be played and one is defined as having in/out points of 1000 and 4600 respectively, while the other has in/out points of 4400 and 7000 should the segment be played without transition or should the segment play the piece of video between 4400 and 4600 twice. Furthermore, what if an out point,

[1]See Russ Sasnett's Master's thesis, *Reconfigurable Video*, 1986.

greater than the defined in point, is specified? Much to our surprise, the reverse play mode of the disc player results, giving an interesting, though probably undesirable, effect. The inclusion of error prompts is a logical addition.

4.5 Videodisc Agenda

One longstanding debate which should probably be reevaluated at this time is that of divulging a videodisc agenda. Defining the header information to appear on all future videodiscs requires that some important decisions be made. We concur with Sasnett and Davenport as to the need to establish corallaries to the Index, Table of Contents, and Dewey Decimal System for video.[1] Whether it is preferable to include indices and tables of content with actual frame numbers on the disc itself at the time of production or add a feature to the system that allows a filmmaker to create these references and store them in the video database is still being examined. From the perspective of the system, whether or not an information frame is included at the beginning of a disc is irrelevant. It is not possible to digitally read the frame and parse any of the data for any type of processing.[2] However, for the user, this seems to be a very useful resource. Since a number of filmmakers often share the capacity of a single videodisc, there should be a simple way to specify at least who shot what, when and where, and the frame range that the footage occupies.

---

[1] See Sasnett.

[2] Actually, there has been progress in the area of reading digital data from optical videodisc by other researcher's in the field. However, one unresolved problem is the format that this data should take. There are more than ten formats and accompanying decoders currently on the market. Once a standard has been established, it is likely that we will want to pursue a scheme that reads such data directly.

4.6 Database

Let us now examine some ways in which the database representation of video material might be improved. One enhancement that will be necessary entails the creation of a *master shot list* of the material on a given videodisc. This would be the responsibility of the person who knows the most about the footage, the filmmaker. In addition to a database record for each segment, a representative icon of the segment should be generated as well. Once defined, copies could be made of the database table containing this master list for anyone who wished to define new segments or slightly change the in/out points of those defined in the master list for their particular application. This is perceived as a being first step in generating meaningful video courseware.

Related to the issue of providing a master list is the question of security that different users have over different tables. At present, no levels of security have been established. The addition of access criteria to particular tables based on some password should be possible by a slight modification to the code that comprises the database manager. This would probably take the form of an initialization routine where a new user of the system is asked to choose a password and a table name. From then on, the password would be the identifier of which table the user has permission to modify.

Another planned change is a way to allow the experienced Ingres user the ability to execute operations on the database not supported by the workstation. This option would allow the user to type a QUEL command just as he would if executing a command at the Ingres command interpreter. The text string would then be sent to the interpreter to be executed, without the need to exit the main workstation program.

Finally, several modifications to the field description of the database have been proposed. One suggested change is the way in which comments are defined. Currently, the user is limited to a text string of no more than twenty characters in the comment field. An alternative to this arrangement is the maintenance of an index in the comment field which would reference a second table devoted to the storage of much longer comments. The addition of dialogue and subject fields has also been recommended. A dialogue seems appropriate for two reasons. The first is that the dialogue in a piece of video is occasionally more important to a particular viewer than the way in which the segment was shot(This is often true in documentary film). Therefore, ability to generate a transcript of a particular scene would be extremely useful. The second reason for advocating the inclusion of dialogue in the database representation has to do with display format. The notion of displaying dialogue on a monitor simultaneously with the video seems particularly suited to educational applications. An obvious one is that of providing subtitles for films in other languages for students studying that particular language.

Chapter 5 Conclusions 27

A system has been described that supplements contemporary approaches to the computer control of video editing systems. It is felt that existing systems are too rigid in their configuration and somewhat inhibiting to the creative process of the film maker. While the workstation configuration is relatively well-defined, the structure is such that enhancement and the introduction of new devices can be easily incorporated into the system. Instrumental in the design of a more flexible medium was the definition of an alternative representation of motion picture material. By removing the cumbersome format of EDLs from the editing process, it is hoped that a greater amount of pure image association may take place. In addition to more meaningful ways of association, the alternative representation makes it possible to easily construct different viewing experiences with the same set of video materials. Thus, while it is possible to view the system as a new tool, it is more accurately depicted as an environment in which new tools may be created.

Appendices 28

This page has intentionally been left blank

A Ingres Database Summary 29

A detailed description of the operations and capabilities supported by the RTI Ingres Database package fills several volumes. However, the Ingres utilities that were used in the development of the database manager of the Video Workstation are representative of a minimal set of functions required for any database application. Therefore, it seems appropriate to describe this set of fundamental operations in the context of the implementation of a database representation of video material. The following discussion outlines the capabilities that currently are supported by the workstation as well as several that we feel are logical additions to the core set of services.

A.1 Database Creation

The creation of a database is one of the few Ingres operations that must be executed at the operating system level. All other utilities(QUEL commands) may be executed either from within one of the available interfaces or from within an application program. The database used in our development was created via the *createdb* command as follows:

createdb videofile

Once a database has been created, individual tables may be declared to partition the data further. Fig. 4 illustrates the hierarchical structure of the videofile database.

A.2 Tables

A single table was created in our implementation and was given the name, *segments* . The segments table was established using the *create* command as follows:

```
create segments(disc_name=c20, seg_name=c20, in_point=i4,
                out_point=i4, theme=c20, character =c50,
                date=i4, comments=c20, icon_address=i4)
```

Each field is declared separately and given a type(integer, text) and a size. Viewed as a two-dimensional row-column structure, each field represents a column while each data record(tuple) represents a row. The complete Ingres description of the table may be viewed by using the help command:

```
help segments
```

The following summary appears in response to the request:

table	:	segments
owner	:	4_d0002
location	:	default
row width	:	146
number of rows	:	6

number of pages : 1
journaling : disabled
storage structure : heap
table type : user table

column name	type	length
disc_name	c	20
seg_name	c	20
in_point	i	4
out_point	i	4
theme	c	20
character	c	50
date	i	4
comments	c	20
icon_address	i	4

The owner of a table is always the login of the user who created the database.[1] .Parameters such as table type and storage structure may be set by the user. The segments table uses the default parameter settings.

A.3 Loading Data

Loading data into a table can be done in several ways. The two most common commands to do this are copy and append. Copy allows data to be transferred from a text file to a table. This has the virtue of adding more than one record to a table with a single invocation of the command. Append adds one new record to the table and requires the data to be specified in the body of the command. An example using the append command is given below:

```
append to segments(disc_nam="Annie Hall",seg_name="Cocaine",
                   in_point=12000, out_point=13450)
```

[1]For our implementation, a development account(4_d0002) on the Project Athena machine, Aphrodite, was used. This is where the Ingres data is currently stored.

A.4 Data Modification

The two utilities provided for data modification and used by the workstation are the *replace* and *delete* commands. Both commands allow for a qualifying statement to specify some range of tuples that will undergo the specified modification. Two examples are given below.

```
delete segments where segments.in_point > 10000

replace segments(in_point = 10*in_point)
                where segments.seg_name="Cocaine"
```

A.5 Queries

The final utility needed to provide the basic tools for a database manager is one which allows the processing of queries. Ingres provides this capability with the *retrieve* command. This command differs from those previously described in that the result of a query is a set of data satisfying some qualifier rather than a self-contained deletion or addition to an existing table. An example using the retrieve command is given below.

```
retrieve segments(segments.disc_name, segments.seg_name)
                where segments.seg_name = "cocaine"
```

The tuples in a table that satisfy the qualifying *where* statement in the query are normally displayed on the monitor being used. GDB(See Appendix B) provides a means for redirecting the data to a local data structure rather than to the screen. One additional feature of the *retrieve* command is that the

data resulting from a query may be redirected to another table given the table name. Thus the ability to copy a master shot-list to a user table is a straightforward extension of the query.

The issues of multiple users and table security require a more thorough look at the structure of Ingres databases. The implementation herein described does not incorporate such provisions. Modifications to the user interface will be required to allow users of the workstation to specify different table names as well as whether a particular user has authority to update a given table. It is hoped that future enhancements will resolve these issues.

B GDB Summary 34

Having established the need for a permanent memory storage environment that has a relational database as its foundation, the link between application program and data will now be explored. The decision to incorporate the capabilities of the RTI Ingres database available through Project Athena presented a major problem. Access to the data stored on an Ingres database has been traditionally limited to an application program that runs on the same machine. Although access is possible either directly using the Ingres Menu System or from within an application program(written in either Fortran or C), no provisions were made for the execution of these accessing modes from a remote location. The main usage and role that GDB[1] plays is allowing a program(in this case a C program) running on machine in a Berkeley Unix[2] environment to create alter and query the relations in an RTI Ingres database stored on some other machine in the network. In accomplishing this end, the GDB software package handles all of the communication protocols inherent in the tranferring of information between machines as well as any inconsistencies between data types.

One additional issue remained to be resolved before work in the creation of a database interface using GDB could proceed.The problem lay in the existence of two versions of the Unix Operating System. While project

---

[1]GDB is an acronym for Global Database. It is a set of C library routines written by Noah Mendelsohn. It is hoped that this very useful set of functions will receive official sanction from Project Athena.

[2]Unix is a trademark of AT&T Bell Laboratories.

Athena is a Berkeley Unix environment, The Hewlett Packard Bobcat computer currently provides the AT&T System V version of Unix. Although Unix is extremely portable in many respects, some of the communication protocals provided in a Berkeley environment are not provided in System V. Thus, the inclusion of two libraries as well as some subtle modifications to the code comprising GDB needed to be made in order to develop and execute the additional software required by the Video Workstation. A discussion of the structure of GDB and the utilities that it provides follows. The details for implementing GDB on a Bobcat will then be presented.

B.1 Building the Bridge

The Unix library routines that comprise GDB offer a variety of asynchronous communication services. The realization of our design specifications was possible through the use of some of the more straightforward GDB tools. Fig. 5 illustrates the link provided by GDB between the database manager and the data stored on the remote Ingres database. It is important to reiterate that once a database has been created on a remote machine, any operation provided by Ingres may be performed within a C program by invoking a function provided by the GDB library. A brute force method to accomplish this end would have been to create a unique routine for each possible operation provided by Ingres. The approach used by GDB is a much more generic one and reduces the programming overhead to the use of three primary GDB utilities. The first of these utilities establishes a connection to a particular database on a remote machine. Moreover, it creates a local identifier for future operations to this database

since simultaneous connections to multiple databases are supported. The syntax of the function is as follows:

```
access_db(db_identifier, &db_handle)
string db_identifier;
DATABASE db_handle;
```

An example of the use of this function is given below.

```
access_db("videofile@aphrodite", &videofile);
```

As stated above this routine maybe used repeatedly if access to more than one database is desired. Once a connection to a remote database has been successfully established, we may now perform queries and operations that modify the database from within our application program. In the example above, an Ingres database(videofile) which resides on the network machine, aphrodite, has been specified as the target database for establishing a connection. The data variable, "videofile", of type DATABASE(a GDB defined type) is used to specify the database upon which a given operation will be performed. There are two functions which gdb provides in order to perform an operation on a particular database. These are: perform_db_operation and db_query.

```
perform_db_operation(db_handle,request)
DATABASE db_handle;
string request;
```

```
db_query(db_handle, relation, query_string)
DATABASE db_handle;
RELATION relation;
string query_string;
```

The perform_db_operation routine may be used to execute any Ingres command with the exception of the retrieve command. An example using the definition of the *segments* table described previously might be:

```
command = "append to segments(disc_name = \"Godfather\",
            seg_name = \"trigger's demise\", inpoint = 3000,
            outpoint=4500,comments=\"don't mess with the mob\")";

perform_db_operation(videofile,command);
```

This would add a new record to the videofile database with the chosen fields taking on the specified values. The result of this operation is merely an integer return code stating whether the operation was successful or not. The issue of performing a database query will now be addressed.

When Ingres is used to retrieve data from a relational database according to some specified criteria, the result of the query is referred to as a *relation*. A relation may be comprised of 0 or more *tuples* (or records) where each of the tuples consists of *fields*. The fact that a database query(accomplished in Ingres by using the retrieve command) returns a set of data and not just an integer return code is why a separate function is needed to provide the programmer with query capability. Thus a local data structure must be provided to store the result of the query(GDB provides the

datatype RELATION to do just that). An example of a query specified at the Ingres command shell would be:

```
retrieve(disc_name = segments.disc_name, segment_name =
        segments.seg_name,in_point = segments.in_point )
        where segments.in_point > 10000
```

Thus, the three fields, disc_name, seg_name and in_point would be returned for each tuple in the database that satisfied the query criteria(having an inpoint in excess of 10000). The identical query would be accomplished in a C program using the db_query routine as follows:

```
db_query(videofile,retrieved_data,
        "(>*disc_name*< = segments.disc_name,
            >*seg_name*< = segments.seg_name,
            >*in_point*< = segments.in_point)
        where segments.in_point > 10000");
```

Once completed, the resulting data will be stored in the *RELATION* , retrieved_data, and may be accessed via additional gdb utilities and used in the body of the main program.

Now that a general description of some of the GDB library routines has been given, it is appropriate to touch briefly on the Ingres/GDB interface at the server site. Fig. 6 illustrates the two major compents of GDB. Once the desired Ingres operation has been specified by the user, the command string

is sent over the network to the remote machine. In order to interpret a given instruction GDB has a command interpreter which resides on the machine where the Ingres database is located. This program(dbserv) receives the command string, parses it, and then send the properly formatted instruction to the Ingres command shell for execution. Therefore, unless this program is running, database accesses will be impossible.[1].A problem with the syntax of the command or a reference to a non-existing table will result in an error that will be communicated back across the network to the application program that attempted the operation.

B.2 Bobcat Implementation

Currently, Project Athena is distributing all the files needed to run GDB on any Berkeley Unix machine in the network. In order to achieve the same level of functionality on an HP Bobcat, follow the procedures described below.

1. After getting the tar file from Project Athena(or the author of the libraries), unpack them in a clean directory.
2. Change the file, **gdb.h,** so that the file, time.h, is looked for in /usr/include, rather than /usr/include/sys. If not available, bring this file over from a Berkeley Unix machine.
3. The files : **socket.h, signal.h, wait.h** must be ported over from a Berkeley machine into the local directory that has the GDB files. A version of **wait.h**

[1]This program should be executed at the operating system level. It may be run in the background by typing:
dbserv &.

does not exist on the Bobcat, while the other two are merely lacking some definitions. Once the first three steps have been completed you can create the library libgdb.a using the Makefile provided by executing, *make libgdb.a* .

4. Once libgdb.a has been created, an application program can only be successfully compiled if two additional libraries are included in the compile statement. An example is:

cc sample.c libgdb.a -lbsd -lbsdipc

These additional libraries provide the additional communication prorocals provided on a Berkeley machine that GDB uses to provide network server capability.

5. Steps 1-4 allow you to successfully compile an application program. Access to an Ingres database will require one last change. The file, **gdb_conn.c**, requires that the declaration:

int *port;

be changed to

u_short *port;

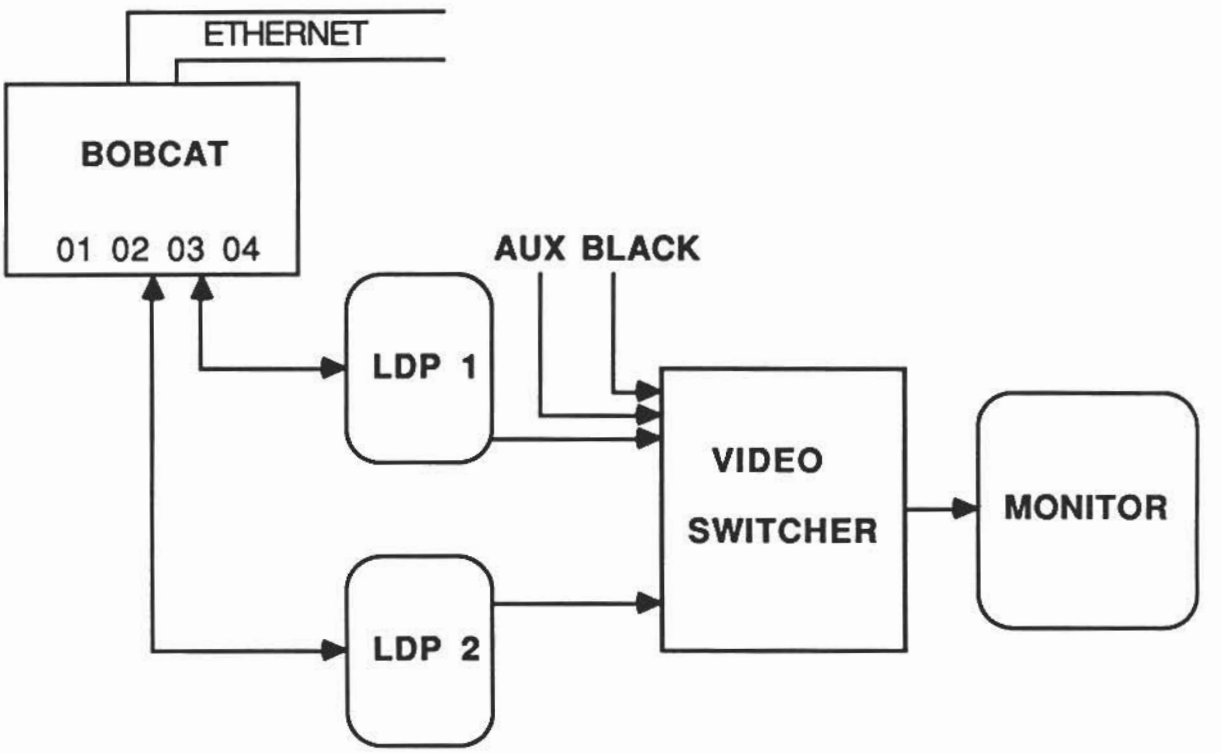

Figure 1: Hardware Configuration

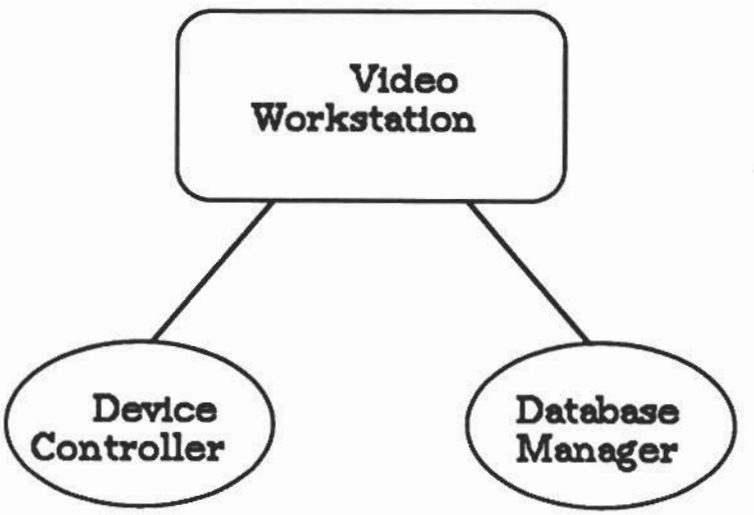

Figure 2: System With Database Manager

Event	Device	A/V	Trans	Dur	Source In	Source Out	Record In	Record Out
001	001	B	C		01:28:54:19	01:29:11:28	01:01:00:00	01::01:17:09
001	002	B	D	01:00	01:27:11:15	01:27:14:05	01:01:17:09	01::01:19:29
002	002	B	C		01:28:17:01	01:28:26:07	01:01:19:29	01::01:29:05
003	aux	B	C		01:26:25:24	01:26:27:12	01:01:29:05	01::01:30:23
004	002	A2V	C		01:26:40:21	01:26:49:29	01:01:30:23	01::01:40:01
005	001	A1	C		01:25:00:09	01:25:21:17	01:01:40:01	01::02:01:09
006	002	A1	C		01:29:34:05	01:29:59:29	01:02:01:09	01::02:27:03
007	001	V	C		01:25:00:09	01:25:21:17	01:01:40:01	01::02:01:09
007	002	V	W000	01:00	01:29:34:05	01:29:59:29	01:02:01:09	01::02:27:03

Figure 3: CMX Edit Decision List

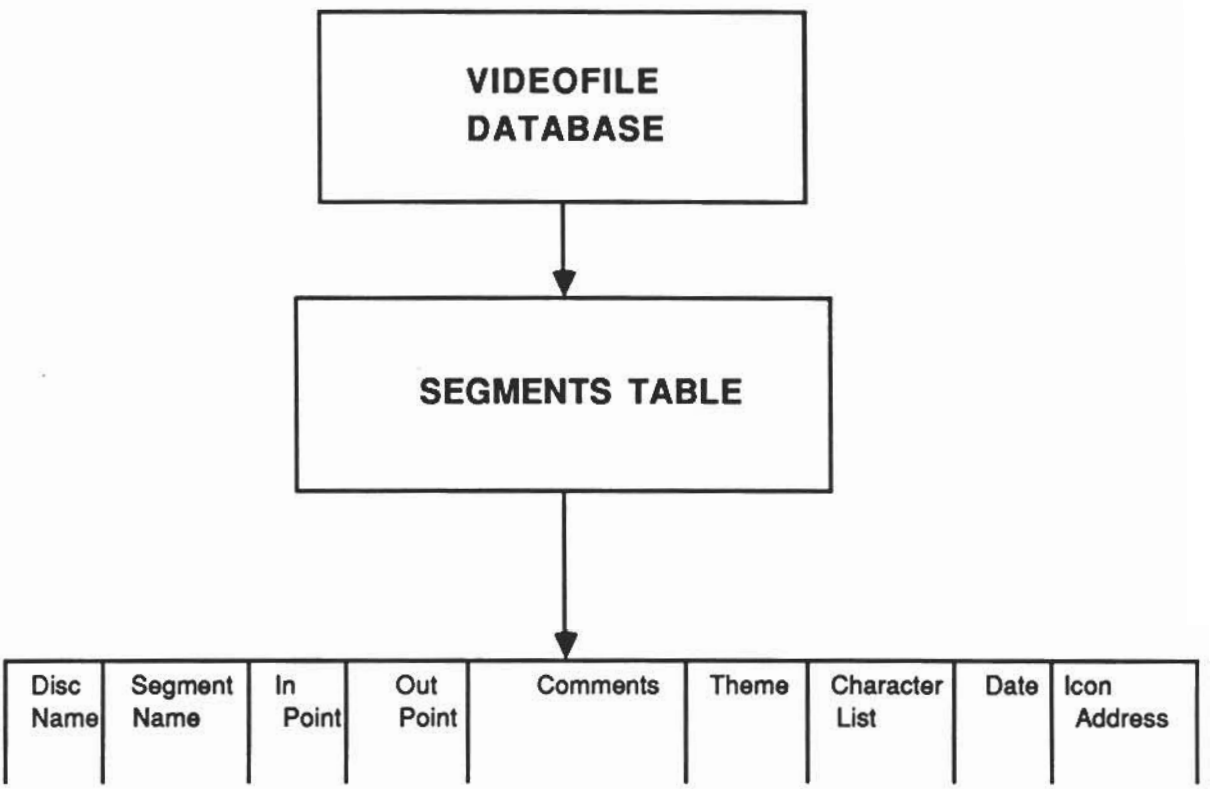

Figure 4: Hierarchical Database Structure

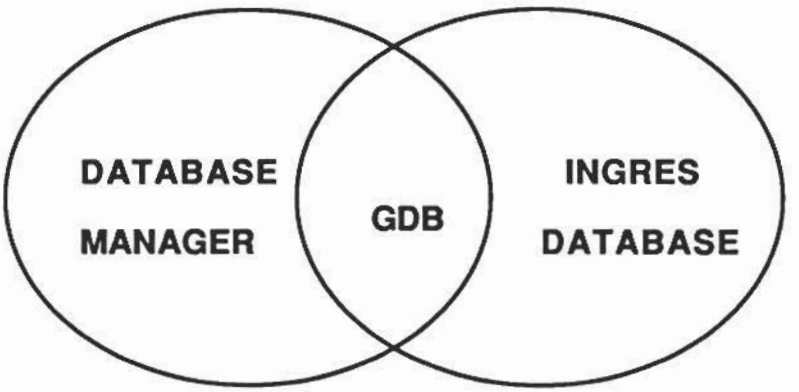

Figure 5: GDB Interface

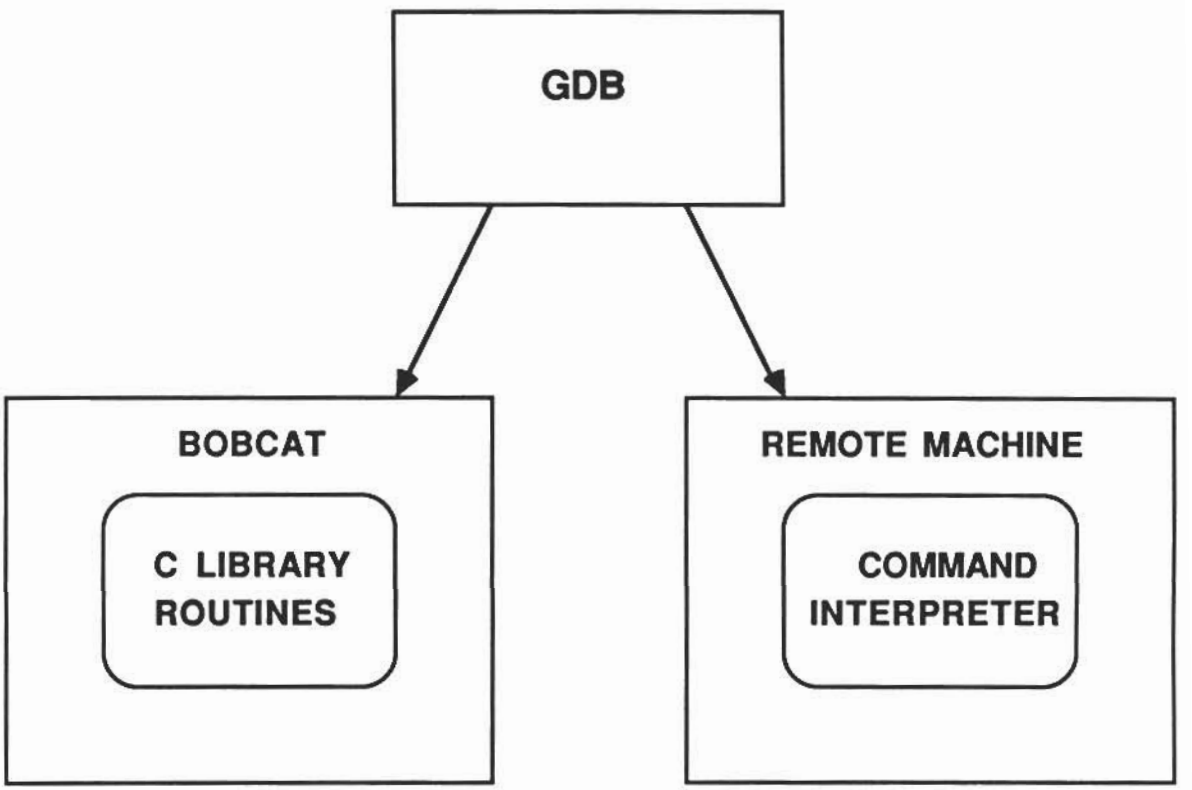

Figure 6: GDB Structure

D C Programs 45

The following C programs were written in order to implement the database manager of the video workstation. The first(dbfuncs.h) is a header file containing definitions of variables used by the primary program(dbfuncs.c).

(dbfuncs.h)

```
/*******************************

 Header file for dbfuncs()

*******************************/

#include "gened.h"   /* for some things */

/*
The following field declarations reflect the current ingres
database definition of the segment database and are used to
create a tuple_descriptor.
*/

char *field_names[] = {"disc_name",
                       "seg_name",
                       "in_point",
                       "out_point",
                       "theme",
                       "character",
                       "date",
                       "comments",
                       "icon_address"};

FIELD_TYPE field_types[] = {STRING_T,     /* disc_name */
                            STRING_T,     /* seg_name */
                            INTEGER_T,    /* in_point */
                            INTEGER_T,    /* out_point */
                            STRING_T,     /* theme */
                            STRING_T,     /* character */
                            INTEGER_T,    /* date */
                            STRING_T,     /* comments,changeto
                                             integer */
                                          /* should reference
                                             longer comment
                                             in another table
                           INTEGER_T};    /*icon_address */


/*
This structure is not currently used but might be useful in
the future.
*/


STRUCTURE struct REL_DESC
{ RELATION retrieved_data;
  TUPLE_DESCRIPTOR tuple_desc;
};

typedef struct REL_DESC *PREL_DESC; /* pointer to REL_DESC */
```

(dbfuncs.h)

```
/*
String Endings often used in the formatting of a quel command
for ingres
*/

char ending[] = "\"";
char ending2[] = ",";
char ending2a[] = "\",";
char ending3[] = ")";
char and_ending[] = " and ";

/*
Declarations required for creating a new record in the
segments database.The values for each field are user-defined
while the actual fields definitions have been pre-defined.
*/

char a_disc_string[50] = "disc_name=\"";
char a_seg_string[50] = "seg_name=\"";
char a_inpoint_string[50] = "in_point=";
char a_outpoint_string[50] = "out_point=";
char a_theme_string[50] = "theme=\"";
char a_character_string[70] = "character=\"";
char a_date_string[50] = "date=";
char a_comments_string[50] = "comments=\"";

char a_disc_string2[20];
char a_seg_string2[20];
char a_inpoint_string2[20];
char a_outpoint_string2[20];
char a_theme_string2[20];
char a_character_string2[50];
char a_date_string2[20];
char a_comments_string2[20];

char quel_append[500] = "append to segments (";

char appquery[] = "append to segments (disc_name=\"NO\",
seg_name=\"seg1\",in_point = 1, out_point = 10,
comments=\"none\", theme=\"none\")";

/*
Declarations for updating an existing record in the segments
database. The values for each field are user-defined while
the actual fields definitions have been pre-defined.
*/

char r_disc_string[50] = "disc_name=\"";
char r_seg_string[50] = "seg_name=\"";
char r_inpoint_string[50] = "in_point=";
char r_outpoint_string[50] = "out_point=";
char r_theme_string[50] = "theme=\"";
char r_character_string[70] = "character=\"";
```

(dbfuncs.h)

```
char r_date_string[50] = "date=";
char r_comments_string[50] = "comments=\"";

char r_disc_string2[20];
char r_seg_string2[20];
char r_inpoint_string2[20];
char r_outpoint_string2[20];
char r_theme_string2[20];
char r_character_string2[50];
char r_date_string2[20];
char r_comments_string2[20];

char quel_replace[500] = "replace s (";
char quel_replace2[30] = " where s.disc_name = \"";
char quel_replace3[30] = " and s.seg_name = \"";


/*
 Declarations for deleting a record from the database
*/

char quel_delete[300] = "delete s where ";
char quel_delete2[30] = "s.disc_name=\"";
char quel_delete3[30] = "s.seg_name=\"";

char d_usr_disc[20];
char d_usr_seg[20];

/*
This string will be used to retrieve all data from the
segments database at the beginning of execution. All fields
for each tuple will be retrieved with no qualification.
*/

char startup_query[] = "(>*disc_name*< =
segments.disc_name,>*seg_name*< =
segments.seg_name,>*in_point*< =
segments.in_point,>*out_point*< = segments.out_point,
>*theme*< = segments.theme,>*character*< =
segments.character,>*date*< = segments.date,>*comments*< =
segments.comments,>*icon_address*< = segments.icon_address)";

/*
The following string is used to set a range variable for the
"segments" table in the videofile database.Range variables
are described in the Ingres Reference Manual.
*/

char range_string[30] = "range of s is segments";


     int  field_count = 9; /* Used when creating the tuple
                              descriptor */
```

(dbfuncs.h)

```
      /*
       * The following defines are for convenience in
addressing
       * the fields in the segments table.
       */

#define DISC_NAME  0
#define SEG_NAME  1
#define IN_POINT   2
#define OUT_POINT  3
#define THEME  4
#define CHARACTER   5
#define DATE 6
#define COMMENTS   7
#define ICON_ADDRESS 8
#define FIELD_LENGTH      21
#define CHARLIST_LENGTH   51
#define FULL_FIELD_LENGTH      50
#define FULL_CHARLIST_LENGTH   70
```

(dbfuncs.c)

```
#define terminatedb(x) /* nothin doin yet */
/*
 *    This program access the videofile database on the
 *     aphrodite machine. A variety of standard ingres
 *     commands are offered to the user in order to append
 *     delete and update data in the segments table of the
 *     videofile database.
 *
 *
 *
 *    Author: D.C. Beauchamp
 */

#include <stdio.h>
#include "gened.h"
#include "gdb.h"
#include "funcs.h"
#define SCOPE extern
#include "global.h"
#include "requests.h"
#include "dbfuncs.h"

extern PKEYREQ getrequest();

RELATION retrieved_data; /* Database contents loaded into
this GDB
                            defined data structure. */
DATABASE video_file;       /* A handle specifying which ingres
                            database will be accessed */

dbfuncs()
{ PKEYREQ keyreq;
  int result;

     /*****************************************************
*****
      *           EXECUTION BEGINS HERE                   *

*************************************************************/

     PRINTF("getting ready for gdb_init()\n");ENDPRINT

keyget:
     printf("\n\nDATABASE> ");
        curkbd = DBKBD_OVERLAY;      /* change meaning of keys
*/
     keyreq = getrequest();          /* get request */
        PRINTF("got request = %d\n",keyreq->request);ENDPRINT

     /***** Do database function **********/

     result = dbdispatch(keyreq->request);
     if(result != QUIT_CODE) goto keyget;
```

(dbfuncs.c)

```
        curkbd = CNTLKBD_OVERLAY;
        return(NOERR);          /* request was to quit, so
return */

      /* END OF DBFUNCS */
}



static int
FUNCTION dbinit()
{
TUPLE_DESCRIPTOR tuple_desc;
int rc1,rc2;

/*
 * Open a connection to the database - identify session as
 * video_file
 */



/**********  INITIALIZE GDB BEFORE USING ANY GDB ROUTINES
*****/

  gdb_init();


/************************************ ******************/

printf("\nEstablishing Connection...");

 if (access_db("videofile@aphrodite", &video_file) !=
DB_OPEN) {
     printf("Cannot connect to video database--giving up\n");
     return(NULL);
}


printf("\nLoading Local Data Structure...");

/*
 * Build the descriptor describing the layout of the tuples
 * to be retrieved, and create an empty relation into which
 * the retrieval will be done.
 */

tuple_desc = create_tuple_descriptor(field_count,
field_names,
                          field_types);
retrieved_data = create_relation(tuple_desc);
```

(dbfuncs.c)

```
/*
 * Do the query for the entire contents of the "segments"
 * table. Put the results in the relation, retrieved_data.
 */



/*
  First step is to load all pertinent ingres data into a
local storage location. All subsequent accesses,thus, will be
faster
*/

rc1 = db_query(video_file, retrieved_data, startup_query);

if(rc1 != OP_SUCCESS) {
     printf("query unsuccesful\n");
     terminatedb(video_file);
     return(NULL);
     }
printf("\nQuery Successful");

/*
Now we define a rangevariable for use in subsequent queries
and database operations.
*/

rc2 = perform_db_operation(video_file,range_string);

if(rc2 != OP_SUCCESS) {
     printf("Range Operation Unsuccesful\n");
     terminatedb(video_file);
     return(NULL);
     }

}


/*
   This is the current function to display a single tuple of
   data from the database.
*/


static FUNCTION print_a_line(tup)
TUPLE     tup;
{
     printf("\n\nDisc Name: %s ",
     STRING_DATA(*(STRING*)(FIELD_FROM_TUPLE(tup,
DISC_NAME))));
     printf("Seg Name: %s ",
     STRING_DATA(*(STRING *)(FIELD_FROM_TUPLE(tup,
SEG_NAME))));
```

(dbfuncs.c)

```
     printf("In_point: %d\n",
     *(int *)(FIELD_FROM_TUPLE(tup, IN_POINT)));
     printf("Out_point: %d\t ",
      *(int *)(FIELD_FROM_TUPLE(tup, OUT_POINT)));
     printf("Theme: %s ",
      STRING_DATA(*(STRING *)(FIELD_FROM_TUPLE(tup,
THEME))));
     printf("Comments: %s\n",
      STRING_DATA(*(STRING *)(FIELD_FROM_TUPLE(tup,
COMMENTS))));
     printf("Date: %d\t",
      STRING_DATA(*(STRING *)(FIELD_FROM_TUPLE(tup, DATE))));
     printf("Characters: %s ",
      STRING_DATA(*(STRING *)(FIELD_FROM_TUPLE(tup,
CHARACTER))));

     /* printf("icon_address: %d\n\n  ",
      *(int *)(FIELD_FROM_TUPLE(tup, ICON_ADDRESS))); */
}



static FUNCTION print_options()
{
     printf("\nDatabase Options\n");
     printf("\n?\t\t\t: List Database Options");
     printf("\ni\t\t\t: Initialize GDB and Load Local Data
             Structure");
     printf("\nl\t\t\t: List Database Contents by Category");
     printf("\na\t\t\t: Add New Record to Database");
     printf("\nu\t\t\t: Update Existing Record in Database");
     printf("\nx\t\t\t: Delete Existing Record From
Database");
     printf("\nq\t\t\t: Exit Database Mode");
}

/*
This function is used by the dbappend to cleanup all the
data pertinent to the last append operation perform. All
relevant
variables are initialized.
*/


static FUNCTION cleanup()
{
int i;

for(i=0; i < FULL_CHARLIST_LENGTH; i++)
{
a_character_string[i] = 0;
}
```

(dbfuncs.c)

```
for(i=0; i < FULL_FIELD_LENGTH; i++)
{
a_disc_string[i] = 0;
a_seg_string[i] = 0;
a_inpoint_string[i] = 0;
a_outpoint_string[i] = 0;
a_comments_string[i] = 0;
a_theme_string[i] = 0;
a_date_string[i] = 0;
}

for(i=0; i < FIELD_LENGTH - 1; i++)
{
a_disc_string2[i] = 0;
a_seg_string2[i] = 0;
a_inpoint_string2[i] = 0;
a_outpoint_string2[i] = 0;
a_comments_string2[i] = 0;
a_theme_string2[i] = 0;
a_character_string2[i] = 0;
a_date_string2[i] = 0;
}

for(i=0; i <500; i++)
{
quel_append[i] = 0;
}

strcpy(a_disc_string,"disc_name=\"");
strcpy(a_seg_string,"seg_name=\"");
strcpy(a_inpoint_string,"in_point=");
strcpy(a_outpoint_string,"out_point=");
strcpy(a_comments_string,"comments=\"");
strcpy(a_theme_string,"theme=\"");
strcpy(a_character_string,"character=\"");
strcpy(quel_append,"append to segments (");
strcpy(a_date_string,"date=");

}

/*
Performs same function as cleanup() for the dbupdate()
function
*/

static FUNCTION u_cleanup()
{
int i;

for(i=0; i < FULL_CHARLIST_LENGTH; i++)
{
r_character_string[i] = 0;
}
```

(dbfuncs.c)

```
for(i=0; i < FULL_FIELD_LENGTH; i++)
{
r_disc_string[i] = 0;
r_seg_string[i] = 0;
r_inpoint_string[i] = 0;
r_outpoint_string[i] = 0;
r_comments_string[i] = 0;
r_theme_string[i] = 0;
r_date_string[i] = 0;
}

for(i=0; i < FIELD_LENGTH -1; i++)
{
r_disc_string2[i] = 0;
r_seg_string2[i] = 0;
r_inpoint_string2[i] = 0;
r_outpoint_string2[i] = 0;
r_comments_string2[i] = 0;
r_character_string2[i] = 0;
r_date_string2[i] = 0;
r_theme_string2[i] = 0;
}

for(i=0; i < 500; i++)
{
quel_replace[i] = 0;
}

strcpy(r_disc_string,"disc_name=\"");
strcpy(r_seg_string,"seg_name=\"");
strcpy(r_inpoint_string,"in_point=");
strcpy(r_outpoint_string,"out_point=");
strcpy(r_comments_string,"comments=\"");
strcpy(r_theme_string,"theme=\"");
strcpy(r_character_string,"character=\"");
strcpy(r_date_string,"date=");
strcpy(quel_replace,"replace s (");
}


/*
Performs same function as cleanup() for the dbdelete()
function.
*/

static FUNCTION d_cleanup()
{
int i;
for(i=0; i <300; i++)
{
quel_delete[i] = 0;
```

(dbfuncs.c)

```
}
for(i=0; i <30; i++)
{
quel_delete2[i] = 0;
quel_delete3[i] = 0;
}
strcpy(quel_delete,"delete s where ");
strcpy(quel_delete2,"s.disc_name=\"");
strcpy(quel_delete3, "s.seg_name=\"");
}


/*
This function is used to make comparisons between a user-
specified
input and a database field value. Spaces must be added to the
end of a user-defined string in order to do the comparison
successfully.
*/

static FUNCTION pad(str)
char str[];
{
int p_test = 0;
int l;
for(l=0,p_test=0;l < FIELD_LENGTH -1; l++)
     {
     if(str[l] == '\0')
          p_test = 1;
     if(p_test == 1)
          str[l] = ' ';
     }
str[FIELD_LENGTH -1] = 0;
}

/*
See the pad function
*/


static FUNCTION pad2(str2)
char str2[];
{
int p_test2 = 0;
int l;
for(l=0,p_test2=0;l < CHARLIST_LENGTH -1; l++)
{
if(str2[l] == '\0')
          p_test2 = 1;
     if(p_test2 == 1)
          str2[l] = ' ';
     }
str2[CHARLIST_LENGTH -1] = 0;
```

(dbfuncs.c)

```
}

static FUNCTION clearstring(str,size)
char str[];
int size;
{
int x;
for(x=0;x < size; x++)
     str[x] = ' ';
str[size] = 0;
}

/*
   If the database operation requested by the user is
supported,
   the appropriate function will be called and executed.
*/


FUNCTION dbdispatch(request)
int request;
{ int result;


      /*
      Main database options loop
      */


          switch(request)
               {

               case R_DBLIST:       result = list();
                         break;

               case R_QUIT:    result = dbquit();
                         break;

               case R_DBHELP:       result = print_options();
                         break;

               case R_DBAPPEND: result = dbappend();
                              break;

               case R_DBINIT:   result =  dbinit();
                         break;

               case R_DBUPDATE: result = dbupdate();
                         break;

               case R_DBDELETE: result = dbdelete();
                         break;
                  default:  result = ERR_DONT_KNOW;
```

(dbfuncs.c)

```
                }

  return(result);
}

/*
This function provides several ways to look at the data down-
loaded
from the database to the retrieved_data relation
*/


static int FUNCTION list()
{ char
choice,l_usr_string[FIELD_LENGTH],char_string[CHARLIST_LENGTH
];
  char *c_string;
  TUPLE q;
  int result, got_one, match, g, l, p;
  result = NOERR;

    printf("\nLIST\n\n");
    printf("Type 'a' for a complete listing of all segments
in the
            database");
    printf("\nType 'd' to list by disc name");
    printf("\nType 't' to list by theme");
    printf("\nType 'c' to list by character\n\nChoice:");
    choice = getachar();
    if(choice  == 'a')
    {
    for (q = FIRST_TUPLE_IN_RELATION(retrieved_data); q!=
NULL;
    q = NEXT_TUPLE_IN_RELATION(retrieved_data,q))
       print_a_line(q);
    }

    if(choice == 'd')
    {
    l_usr_string[FIELD_LENGTH - 1] = 0;
    clearstring(l_usr_string,FIELD_LENGTH - 1);
    printf("\nDisc Name: ");
    getchar();
    scanf("%[^\n]",l_usr_string);
    pad(l_usr_string);
     if ( (result =
        strfieldmatch(
            FIRST_TUPLE_IN_RELATION(retrieved_data)
            ,DISC_NAME,l_usr_string)) < 0 )
          result = ERR;
    }
```

(dbfuncs.c)

```
    if(choice == 'c')
    {
    clearstring(char_string,CHARLIST_LENGTH - 1);
    printf("\nCharacter: ");
    getchar();
    scanf("%[^\n]",char_string);

for(match=0,got_one=0,q=FIRST_TUPLE_IN_RELATION(retrieved_dat
a);
       q != NULL; q =
NEXT_TUPLE_IN_RELATION(retrieved_data,q))
     {
      c_string = STRING_DATA(* (STRING *)
                 (FIELD_FROM_TUPLE(q,CHARACTER)));

      for(g=0; g < CHARLIST_LENGTH - 1; g++)
            {
            if(match == 1)
                  {
                  got_one = 1;
                  match = 0;
                  break;
                  }
            if(char_string[0] == c_string[g])
                  {
                  for(l=1,p=g+1;p < CHARLIST_LENGTH - 1;l++,p++)
                        {
                        if(char_string[l] == '\0')
                              {
                              print_a_line(q);
                              match = 1;
                              break;
                              }
                        if(char_string[l] == c_string[p])
                              ;
                        else
                              break;
                        }
                  }
            }
      }
      if(got_one != 1)
            printf("\nNO RECORDS FOUND");
}


    if(choice == 't')
    {
    l_usr_string[FIELD_LENGTH - 1] = 0;
    clearstring(l_usr_string,FIELD_LENGTH - 1);
    printf("\nSearch Theme: ");
    getchar();
    scanf("%[^\n]",l_usr_string);
```

(dbfuncs.c)

```
    pad(l_usr_string);
    if ( (result = strfieldmatch
                    (FIRST_TUPLE_IN_RELATION(retrieved_data),
                    THEME,
                    l_usr_string)) < 0 )

          result = ERR;
    else
                result = NOERR;
    }

return(result);

}    /* end of list() */


/*
This function returns control to main where the device
drivers
may be accessed directly.
*/


static FUNCTION dbquit()
{
    printf("\nEXITING DATABASE MODE\n");

/* terminatedb does not actually do anything at this point */

    terminatedb(video_file);
    return(QUIT_CODE);
}
          /* end of quit */


/*
This function allows a user to specify data to be entered
into the database as a new record(tuple).
*/


static FUNCTION dbappend()
{ TUPLE a_tup;

char   a_local_comments[FIELD_LENGTH],
       a_local_seg[FIELD_LENGTH],
       answer1[],
       a_local_theme[FIELD_LENGTH],
       a_local_character[CHARLIST_LENGTH],
       a_local_disc[FIELD_LENGTH];
int   a_inpoint_int,
      result,
      g,
```

(dbfuncs.c)

```
      rc2,
      a_outpoint_int,
      a_date_int;
TUPLE_DESCRIPTOR tuple_desc3;


tuple_desc3 = create_tuple_descriptor(field_count,
field_names,
                          field_types);
  cleanup();
  a_tup = create_tuple(tuple_desc3); /* For local append to
the
                                     retrieved_data relation
*/
  initialize_tuple(a_tup);
  printf("\n\nAPPEND FUNCTION");
  printf("\nYou Must enter both a Disc Name and Segment Name
for       each record you create.\nAll other fields are
optional.
          To leave a field blank hit RETURN");

  printf("\n\nDisc Name: ");
  getchar();
  scanf("%[^\n]", a_disc_string2);
  if(a_disc_string2[0] == ABORTCHAR)          /* abort? */
  return(0);
 if(a_disc_string2[0] != 0)
     {
      strcpy(a_local_disc,a_disc_string2);
      pad(a_local_disc);

string_alloc((STRING*)FIELD_FROM_TUPLE(a_tup,DISC_NAME),
                    FIELD_LENGTH);
      strcpy(STRING_DATA(*((STRING*)
             FIELD_FROM_TUPLE(a_tup,DISC_NAME))),
          a_local_disc);
      strcat(a_disc_string, a_disc_string2);
      strcat(a_disc_string,ending2a);
      strcat(quel_append,a_disc_string);
      /* printf("\nhope = %s",quel_append); */
     }

  printf("\nSegment Name: ");
  getchar();
  scanf("%[^\n]", a_seg_string2);
  if(a_seg_string2[0] == ABORTCHAR)
  return(0);
  if (a_seg_string2[0] != 0)
      {
      strcpy(a_local_seg,a_seg_string2);
      pad(a_local_seg);
      string_alloc((STRING*)FIELD_FROM_TUPLE(a_tup,SEG_NAME)
                  ,FIELD_LENGTH);
```

(dbfuncs.c)

```
      strcpy(STRING_DATA(*((STRING*)
             FIELD_FROM_TUPLE(a_tup,SEG_NAME))),a_local_seg);
      strcat(a_seg_string, a_seg_string2);
      strcat(a_seg_string,ending2a);
      strcat(quel_append,a_seg_string);
      /* printf("\nhope = %s",quel_append); */
     }

  if ( ef2 & INPT_BIT )
      strcpy(a_inpoint_string2,inframe); /* get currently
marked
                                        in point
                                    */
  printf("\nIn Point: [%s] ",a_inpoint_string2);
  getchar();
  scanf("%[^\n]", a_inpoint_string2);
  if(a_inpoint_string2[0] == ABORTCHAR)
  return(0);
  if (a_inpoint_string2[0] != 0)
     {a_inpoint_int = atol(a_inpoint_string2);
      *(int *)FIELD_FROM_TUPLE(a_tup,IN_POINT) =
a_inpoint_int;
      strcat(a_inpoint_string, a_inpoint_string2);
      strcat(a_inpoint_string,ending2);
      strcat(quel_append,a_inpoint_string);
      /* printf("\nhope = %s",quel_append); */
     }

  if ( ef2 & OUTPT_BIT )
      strcpy(a_outpoint_string2,outframe);     /* get
currently                                          marked in
point */

  printf("\nOut Point: [%s] ",a_outpoint_string2);
  getchar();
  scanf("%[^\n]", a_outpoint_string2);
  if(a_outpoint_string2[0] == '^a')
  return(0);
  if (a_outpoint_string2[0] != 0)
     {a_outpoint_int = atol(a_outpoint_string2);
      *(int *)FIELD_FROM_TUPLE(a_tup,OUT_POINT) =
a_outpoint_int;
      strcat(a_outpoint_string, a_outpoint_string2);
      strcat(a_outpoint_string,ending2);
      strcat(quel_append,a_outpoint_string);
      /* printf("\nhope = %s",quel_append); */
     }

  printf("\nComments: ");
  getchar();
  scanf("%[^\n]", a_comments_string2);
  if(a_comments_string2[0] == ABORTCHAR)
     return(0);
```

(dbfuncs.c)

```
  if (a_comments_string2[0] != 0)
     {
      strcpy(a_local_comments,a_comments_string2);
      pad(a_local_comments);
      string_alloc((STRING*)FIELD_FROM_TUPLE(a_tup,COMMENTS),
                   FIELD_LENGTH);
      strcpy(STRING_DATA(*((STRING*)

FIELD_FROM_TUPLE(a_tup,COMMENTS))),a_local_comments);
      strcat(a_comments_string, a_comments_string2);
      strcat(a_comments_string,ending2a);
      strcat(quel_append,a_comments_string);
      /* printf("\nhope = %s",quel_append); */
      }

printf("\nCharacter List: ");
  getchar();
  scanf("%[^\n]", a_character_string2);
  if (a_character_string2[0] == ABORTCHAR)
     return(0);
  if(a_character_string2[0] != 0)
     {
      strcpy(a_local_character,a_character_string2);
      pad(a_local_character);

string_alloc((STRING*)FIELD_FROM_TUPLE(a_tup,CHARACTER),
                   CHARLIST_LENGTH);
      strcpy(STRING_DATA(*((STRING*)

FIELD_FROM_TUPLE(a_tup,CHARACTER))),a_local_character);
      strcat(a_character_string, a_character_string2);
      strcat(a_character_string,ending2a);
      strcat(quel_append,a_character_string);
      /* printf("\nhope = %s",quel_append); */
     }

  printf("\nTheme: ");
  getchar();
  scanf("%[^\n]", a_theme_string2);
  if(a_theme_string2[0] == ABORTCHAR)
     return(0);
  if(a_theme_string2[0] != 0)
     {
     strcpy(a_local_theme,a_theme_string2);
     pad(a_local_theme);
     string_alloc((STRING*)FIELD_FROM_TUPLE
                 (a_tup,THEME),FIELD_LENGTH);

strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(a_tup,THEME)))
,
           a_local_theme);
     strcat(a_theme_string, a_theme_string2);
     strcat(a_theme_string,ending2a);
```

(dbfuncs.c)

```
      strcat(quel_append,a_theme_string);
      /* printf("\nhope = %s",quel_append); */
      }

   printf("\nDate: ");
   getchar();
   scanf("%[^\n]", a_date_string2);
   if (a_date_string2[0] == ABORTCHAR)
      return(0);
   if(a_date_string2[0] != 0)
      {a_date_int = atol(a_date_string2);
       *(int *)FIELD_FROM_TUPLE(a_tup,DATE) = a_date_int;
       strcat(a_date_string, a_date_string2);
       strcat(a_date_string,ending2);
       strcat(quel_append,a_date_string);
       /* printf("\nhope = %s",quel_append); */
      }

     if ( (result =
         segment_search(
             FIRST_TUPLE_IN_RELATION(retrieved_data)
             ,a_local_disc,a_local_seg)) < 0 )

                      ;  /* if no match go ahead with the
append */
     else
        {
         printf("\n\nSO SORRY. That segment name is already
being
                  used on the disc. Please choose another.");
         result = ERR;
         return(result);
        }


   for(g=0; g < 500; g++)
      {
      if(quel_append[g] == '\0')
           {
           if(quel_append[g -1] == ',')
                     quel_append[g-1] = ')';
           else
                     quel_append[g] = ')';
           break;
           }
      }

   /* printf("\n\nhopeful query: %s\n",quel_append); */


printf("\n\nNew segment defined as: ");
print_a_line(a_tup);
printf("\n\nAre you sure you want to add the segment
```

(dbfuncs.c)

```
        definedabove?");
scanf("%s", answer1);
if(answer1[0] != 'y')
printf("\nAPPEND ABORTED\n");
else
     {
     rc2 = perform_db_operation(video_file,quel_append);
     if(rc2 != OP_SUCCESS) {
     printf("Append Unsuccesful\n");
     return;
     }
else
     {
     ADD_TUPLE_TO_RELATION(retrieved_data,a_tup);
     printf("\nRECORD ADDED TO DATABASE");
     }

}
}  /* end of append */

/*
This function allows an update to be performed on a database
record
*/

static FUNCTION dbupdate()
{
TUPLE r_tup;
TUPLE existing_tup;
TUPLE q;
int rc;
int g;
char u_usr_disc[FIELD_LENGTH];
char u_usr_seg[FIELD_LENGTH];
char up_usr_disc[FIELD_LENGTH];
char up_usr_seg[FIELD_LENGTH];
char *u_rel_disc;
char *u_rel_seg;
int u_exists;
char answer1[];
char r_local_character[CHARLIST_LENGTH];
char r_local_comments[FIELD_LENGTH];
char r_local_seg[FIELD_LENGTH];
char r_local_theme[FIELD_LENGTH];
char r_local_disc[FIELD_LENGTH];
int r_inpoint_int;
int r_outpoint_int;
int r_date_int;
TUPLE_DESCRIPTOR tuple_desc4;

tuple_desc4 = create_tuple_descriptor(field_count,
field_names,
                                      field_types);
```

(dbfuncs.c)

```
u_cleanup();
r_tup = create_tuple(tuple_desc4);
initialize_tuple(r_tup);
printf("\n\nUPDATE FUNCTION");
printf("\n\nTo uniquely specify a video segment for updating,
please
      specify");
printf("\na disc name and a segment name. Use the list(l)
command if
      you");
printf("\ncannot recall the required names. Make sure that a
disc
      does not");
printf("\nhave two segments with the same name. If duplicate
      segments exist,");
printf("\nany updates will affect both records.");

 printf("\n\nDisc Name: ");
 getchar();
 scanf("%[^\n]",u_usr_disc);
 printf("\nSegment Name: ");
 getchar();
 scanf("%[^\n]",u_usr_seg);
 strcpy(up_usr_disc,u_usr_disc);
 strcpy(up_usr_seg,u_usr_seg);
 pad(u_usr_seg);
 pad(u_usr_disc);

 for (u_exists = 0,q =
FIRST_TUPLE_IN_RELATION(retrieved_data); q!= NULL;q =
NEXT_TUPLE_IN_RELATION(retrieved_data,q))
     {
     u_rel_disc  = STRING_DATA(* (STRING *)
                  (FIELD_FROM_TUPLE(q,DISC_NAME)));
     u_rel_seg   = STRING_DATA(* (STRING *)
                  (FIELD_FROM_TUPLE(q,SEG_NAME)));

     if(!strcmp(u_usr_disc,u_rel_disc))
          {
          if(!strcmp(u_usr_seg,u_rel_seg))
          {
           u_exists = 1;
          printf("\n\nYour record currently contains the
following
                  data:");
          print_a_line(q);
          existing_tup = q;
          r_inpoint_int = *(int
*)FIELD_FROM_TUPLE(q,IN_POINT);
          r_outpoint_int = *(int
*)FIELD_FROM_TUPLE(q,OUT_POINT);
          r_date_int = *(int *)FIELD_FROM_TUPLE(q,DATE);
          strcpy(r_local_disc,STRING_DATA(*((STRING*)
```

(dbfuncs.c)

```
                FIELD_FROM_TUPLE(q,DISC_NAME))));
            strcpy(r_local_seg,STRING_DATA(*((STRING*)
                 FIELD_FROM_TUPLE(q,SEG_NAME))));
            strcpy(r_local_comments,STRING_DATA(*((STRING*)
                FIELD_FROM_TUPLE(q,COMMENTS))));
            strcpy(r_local_theme,STRING_DATA(*((STRING*)
                FIELD_FROM_TUPLE(q,THEME))));
            strcpy(r_local_character,STRING_DATA(*((STRING*)
                FIELD_FROM_TUPLE(q,CHARACTER))));
            break;
            }
        }
        }

 if(u_exists != 1)
      {
      printf("\n\nNo Match. Either the specified disc does not
exist
              or the");
      printf("\nsegment doesn't exist on the disc. Check with
the
              list command");
      return(ERR);
      }

printf("\n\nType a new value or hit RETURN if you do not
wish\nto change the current value.");

printf("\n\nDisc Name: ");
getchar();
scanf("%[^\n]", r_disc_string2);
if(r_disc_string2[0] == ABORTCHAR)
      return(0);
if(r_disc_string2[0] != 0)
      {
      strcpy(r_local_disc,r_disc_string2);
      pad(r_local_disc);
      strcat(r_disc_string, r_disc_string2);
      strcat(r_disc_string,ending2a);
      strcat(quel_replace,r_disc_string);
      /* printf("\nhope = %s",quel_replace); */
      }

printf("\n\nSegment Name: ");
getchar();
scanf("%[^\n]", r_seg_string2);
if(r_seg_string2[0] == ABORTCHAR)
      return(0);
if(r_seg_string2[0] != 0)
      {
      strcpy(r_local_seg,r_seg_string2);
      pad(r_local_seg);
      strcat(r_seg_string, r_seg_string2);
```

(dbfuncs.c)

```
      strcat(r_seg_string,ending2a);
      strcat(quel_replace,r_seg_string);
      /* printf("\nhope = %s",quel_replace); */
      }

printf("\n\nInpoint: ");
getchar();
scanf("%[^\n]", r_inpoint_string2);
if(r_inpoint_string2[0] == ABORTCHAR)
      return(0);
if(r_inpoint_string2[0] != 0)
      {
      r_inpoint_int = atol(r_inpoint_string2);
      strcat(r_inpoint_string, r_inpoint_string2);
      strcat(r_inpoint_string,ending2);
      strcat(quel_replace,r_inpoint_string);
      /* printf("\nhope = %s",quel_replace); */
      }

printf("\n\nOutpoint: ");
getchar();
scanf("%[^\n]", r_outpoint_string2);
if(r_outpoint_string2[0] == ABORTCHAR)
      return(0);
if(r_outpoint_string2[0] != 0)
      {
      r_outpoint_int = atol(r_outpoint_string2);
      strcat(r_outpoint_string, r_outpoint_string2);
      strcat(r_outpoint_string,ending2);
      strcat(quel_replace,r_outpoint_string);
      /* printf("\nhope = %s",quel_replace); */
      }

printf("\n\nComments: ");
getchar();
scanf("%[^\n]", r_comments_string2);
if(r_comments_string2[0] == ABORTCHAR)
      return(0);
if(r_comments_string2[0] != 0)
      {
      strcpy(r_local_comments,r_comments_string2);
      pad(r_local_comments);
      strcat(r_comments_string, r_comments_string2);
      strcat(r_comments_string,ending2a);
      strcat(quel_replace,r_comments_string);
      /* printf("\nhope = %s",quel_replace); */
      }

printf("\n\nTheme: ");
getchar();
scanf("%[^\n]", r_theme_string2);
if(r_theme_string2[0] == ABORTCHAR)
      return(0);
```

(dbfuncs.c)

```
if(r_theme_string2[0] != 0)
     {
     strcpy(r_local_theme,r_theme_string2);
     pad(r_local_theme);
     strcat(r_theme_string, r_theme_string2);
     strcat(r_theme_string,ending2a);
     strcat(quel_replace,r_theme_string);
     /* printf("\nhope = %s",quel_replace); */
     }

printf("\n\nCharacter List: ");
getchar();
scanf("%[^\n]", r_character_string2);
if(r_character_string2[0] == ABORTCHAR)
     return(0);
if(r_character_string2[0] != 0)
     {
     strcpy(r_local_character,r_character_string2);
     pad2(r_local_character);
     strcat(r_character_string, r_character_string2);
     strcat(r_character_string,ending2a);
     strcat(quel_replace,r_character_string);
     /* printf("\nhope = %s",quel_replace); */
     }

printf("\n\nDate: ");
getchar();
scanf("%[^\n]", r_date_string2);
if(r_date_string2[0] == ABORTCHAR)
     return(0);
if(r_date_string2[0] != 0)
     {
     r_date_int = atol(r_date_string2);
     strcat(r_date_string, r_date_string2);
     strcat(r_date_string,ending2);
     strcat(quel_replace,r_date_string);
     /* printf("\nhope = %s",quel_replace); */
     }

for(g=0; g < 300; g++)
     {
     if(quel_replace[g] == '\0')
          {
          if(quel_replace[g-1] == ',')
               quel_replace[g-1] = ')';
          else
               quel_replace[g] = ')';

          break;
          }
     }
strcat(quel_replace,quel_replace2);
strcat(quel_replace,up_usr_disc);
```

(dbfuncs.c)

```
strcat(quel_replace,ending);
strcat(quel_replace,quel_replace3);
strcat(quel_replace,up_usr_seg);
strcat(quel_replace,ending);

/* printf("\n\nhope2 = %s",quel_replace); */

string_alloc((STRING*)FIELD_FROM_TUPLE(r_tup,DISC_NAME),
             FIELD_LENGTH);
strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(r_tup,DISC_NAM
E))),
      r_local_disc);
string_alloc((STRING*)FIELD_FROM_TUPLE(r_tup,SEG_NAME),
             FIELD_LENGTH);
strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(r_tup,SEG_NAME
))),
      r_local_seg);
string_alloc((STRING*)FIELD_FROM_TUPLE(r_tup,COMMENTS),
            FIELD_LENGTH);
strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(r_tup,COMMENTS
))),
      r_local_comments);
string_alloc((STRING
*)FIELD_FROM_TUPLE(r_tup,THEME),FIELD_LENGTH);
strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(r_tup,THEME)))
,
      r_local_theme);
string_alloc((STRING*)FIELD_FROM_TUPLE(r_tup,CHARACTER),
             CHARLIST_LENGTH);
strcpy(STRING_DATA(*((STRING*)FIELD_FROM_TUPLE(r_tup,CHARACTE
R))),
      r_local_character);
*(int *)FIELD_FROM_TUPLE(r_tup,IN_POINT) = r_inpoint_int;
*(int *)FIELD_FROM_TUPLE(r_tup,OUT_POINT) = r_outpoint_int;
*(int *)FIELD_FROM_TUPLE(r_tup,DATE) = r_date_int;

printf("\n\nNew definition for segment is: ");
print_a_line(r_tup);

printf("\n\nAre you sure you want to update the segment
\"%s\", on
       the",up_usr_seg);
printf(" disc, \"%s\" ?", up_usr_disc);
scanf("%s", answer1);
if(answer1[0] != 'y')
     printf("\nUPDATE ABORTED\n");

 else
 {
     rc = perform_db_operation(video_file,quel_replace);
     if(rc != OP_SUCCESS) {
          printf("Update Unsuccesful\n");
          return;
```

(dbfuncs.c)

```
        }
        else
                {

        REMOVE_TUPLE_FROM_RELATION(retrieved_data,existing_tup);
                ADD_TUPLE_TO_RELATION(retrieved_data,r_tup);
                printf("\nUPDATE COMPLETED");
                }
 }

}
/* END OF dbupdate() */


/*
This function deletes a tuple from the database on the remote
ingres database as well as in the local retrieved_data
relation
*/

static FUNCTION dbdelete()
{
TUPLE d_kill_tup;
int d_exists = 0;
char d_usr_disc[FIELD_LENGTH];
char answer1[];
char d_usr_seg[FIELD_LENGTH];
char dp_usr_disc[FIELD_LENGTH];
char dp_usr_seg[FIELD_LENGTH];
char *d_rel_disc;
char *d_rel_seg;
int rc;
TUPLE q;

printf("\n\nDELETE FUNCTION");
printf("\n\nTo uniquely specify a video segment for deletion,
please specify");
printf("\na disc name and a segment name. Use the list(l)
command if you");
printf("\ncannot recall the required names.");
d_cleanup();
printf("\n\nDisc Name: ");
getchar();
scanf("%[^\n]",d_usr_disc);
printf("\nSegment Name: ");
getchar();
scanf("%[^\n]",d_usr_seg);
strcpy(dp_usr_disc,d_usr_disc);
strcpy(dp_usr_seg,d_usr_seg);
pad(d_usr_disc);
pad(d_usr_seg);
```

(dbfuncs.c)

```
 for (d_exists = 0,q =
FIRST_TUPLE_IN_RELATION(retrieved_data); q!=
      NULL; q = NEXT_TUPLE_IN_RELATION(retrieved_data,q))
        {
        d_rel_disc = STRING_DATA(*(STRING*)
                     (FIELD_FROM_TUPLE(q,DISC_NAME)));
        d_rel_seg = STRING_DATA(*(STRING*)
                     (FIELD_FROM_TUPLE(q,SEG_NAME)));

     if(!strcmp(d_usr_disc,d_rel_disc))
          {
          if(!strcmp(d_usr_seg,d_rel_seg))
          {
          printf("\n\nYour record currently contains the
following
                  data:");
          print_a_line(q);
          d_exists = 1;
          d_kill_tup = q;
          break;
          }
          }
     }
 if(d_exists != 1)
     {
     printf("\n\nNo Match. Either the specified disc does not
exist
           or the");
     printf("\nsegment doesn't exist on the disc. Check with
the
           list command");
     return(ERR);
     }


printf("\n\nAre you sure you want to delete the segment
\"%s\",
      on the",dp_usr_seg);

printf(" disc, \"%s\" ?", dp_usr_disc);
scanf("%s", answer1);
if(answer1[0] != 'y')
     printf("\nDELETION ABORTED\n");
 else
     {
     strcat(quel_delete2, dp_usr_disc);
     strcat(quel_delete3, dp_usr_seg);
     strcat(quel_delete,quel_delete2);
     strcat(quel_delete,ending);
     strcat(quel_delete,and_ending);
     strcat(quel_delete,quel_delete3);
     strcat(quel_delete,ending);
```

(dbfuncs.c)

```
/* eliminate the return and terminate db commands in these
   error message routines
*/



     rc = perform_db_operation(video_file,quel_delete);
     if(rc != OP_SUCCESS) {
          printf("Deletion Unsuccesful\n");
          return;
          }
     else
          {

     REMOVE_TUPLE_FROM_RELATION(retrieved_data,d_kill_tup);
          printf("\nDELETION COMPLETED");
          }
     }
}

/* END OF dbupdate() */


/*
This function performs a comparison of a user input and a
field
from a tuple in the retrieved_data relation
*/


static int FUNCTION
strfieldmatch(start_tuple,field_index,string)
TUPLE start_tuple;
int field_index;
char *string;
{ int match, result;
  TUPLE q;
  char *d_string;

  for (match = 0,q = start_tuple; q!= NULL;
       q = NEXT_TUPLE_IN_RELATION(retrieved_data,q))

       {
       d_string = STRING_DATA(*(STRING*)
                 (FIELD_FROM_TUPLE(q,field_index)));
        if(!strcmp(string,d_string))
          {
          match = 1;
          print_a_line(q);
          }
       }
```

(dbfuncs.c)

```
  if ( match == 0 )
     {
     printf("\n\nNO RECORDS FOUND.");
     result = ERR;
     }
  else
     result = NOERR;
  return(result);
}


/*
This function checks to see if a tuple with the specified
disc name
and segment name exists in the retrieved_data relation.
*/


static int FUNCTION
segment_search(start_tuple,disc_string,seg_string)
TUPLE start_tuple;
char seg_string[FIELD_LENGTH -1];
char disc_string[FIELD_LENGTH -1];
{
int exists, result;
TUPLE q;
char *cmp_disc;
char *cmp_seg;


 for (exists = 0,q = start_tuple; q!= NULL;
     q = NEXT_TUPLE_IN_RELATION(retrieved_data,q))
     {
     cmp_disc  = STRING_DATA(* (STRING *)
                          (FIELD_FROM_TUPLE(q,DISC_NAME)));
     cmp_seg   = STRING_DATA(* (STRING *)
                (FIELD_FROM_TUPLE(q,SEG_NAME)));
     if(!strcmp(disc_string,cmp_disc))
          {
          if(!strcmp(seg_string,cmp_seg))
               exists = 1;
          }
     }

  if ( exists == 0 ){
     result = ERR;
     }
  else
      result = NOERR;
  return(result);
}
```

References 75

1. Anderson, Gary H. Video Editing and Post Production (New York: Knowledge Industry Publications, Inc., 1984).

2. Claxton, William "PC-TV", PC World (February 1984).
3. Kernigan/Ritchie The C Programming Language (New Jersey: PrenticeHall, Inc., 1984).

4. Lippman, Andy Videodiscs and Optical Storage (MIT Architecture Machine Group).

5. Sasnett, Russ Reconfigurable Video (MIT Master's Thesis 1986).