Homer: A Video Story Generator

by

Lee Hayes Morgenroth

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1992

Author .....
Department of Electrical Engineering and Computer Science
May 18, 1992

Certified by .....
Glorianna Davenport
Assistant Professor of Media Technology
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

Homer: A Video Story Generator

by

Lee Hayes Morgenroth

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1992, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis includes the design, implementation, and use of Homer, a video story generator. Homer takes a specific story model format as input. Based on this model, Homer creates a video story from a supplied database of logged video. Along with the story, a report is created of how well the story model matches the video. The purpose of this application is to encode expert editing knowledge in the form of story models. These models can then be reused by non-experts to create meaningful edits from different collections of video.

Thesis Supervisor: Glorianna Davenport
Title: Assistant Professor of Media Technology

Acknowledgments

I would like to thank my family and Laura for dealing with me throughout this endeavor. I would also like to thank all the members of the Interactive Cinema group. Special thanks to Glorianna Davenport for supporting me through it all.

Contents

List of Figures

Chapter 1

Introduction

Homer is a video story generator. It is not do the job of a human editor. The act of editing is a complex task that is beyond the scope of Homer. An editor must worry about the transitions between each shot in a piece as well as the story being told. An editor must also concern himself with the aesthetics of the story being created. Homer, in the process of creating video sequences, attempts to mimic the story generation process used by human editors.

Homer uses the structure inherent in narratives to create a video story as output. All stories have structure. There are various levels to this structure. On the highest level there is the story purpose. In an adventure story this may be "the hero saves the day." There are of course more specific levels of structure to such a story. The hero may first have to capture the villain, then destroy the doomsday device in order to save the day. Each one of these tasks can be broken down into more specific actions. All actions on all levels together make up the structure of a story.

In film this structure is reflected in the sequences and scenes that make up the movie. Each scene contains one or more changes or actions that advance the story. In a typical love story, each of the future companions has to be introduced. Each of these introductions is accomplished in a scene. Further scenes may develop these and other characters. At some point there will be a scene where the protagonists meet.

Each scene in the movie takes the story closer to its conclusion. Homer allows a user to map out this story structure using description, and creates a video story applying the user's story model as a template.

Homer takes a story model and a database of logged video as input. Homer reads the story model and tries to choose and order shots from the video database using the story model. This narrative should reflect the characteristics outlined in the story model. The specifics of the story model will be discussed in a later section. Basically, a story model must contain a significant amount of knowledge that Homer can use to create a meaningful video sequence.

Homer has multiple uses in modern day video processing. If careful video logs are kept for a project, Homer can be used as an editor's assistant. One of Homer's story models, that describes the story the editor wants to create, can be fed to Homer with the logged video. Homer will provide the editor with a rough cut of one such story. Homer also provides extra footage that could be used at each point in the story. Homer provides easy access to useful footage for the editor to work with. In some cases, Homer creates a true rough cut that only needs finishing touches from the human editor. In either case, work is saved by using Homer. The question of how much time is saved depends on how much time was spent logging, and how much time was spent creating the story model. Issues of video logging will be discussed later in the paper, but if footage is ever to be reused, careful logging is essential. Homer's story model feature is designed for story model reuse. Once a small library of general story models is created, a user should be able to customize an existing model to satisfy most story types.

Homer can also be used as a front end to a large database of logged video. It can be used to search an archive by an editor or Homer can be used for general informational purposes. Homer can create stories about any subject for which there is video and a model available. A slightly modified general story model is a powerful tool for pulling coherent narratives from large video archives. Automated Content Editor (ACE) was

an early example of such an application. ACE formulated news stories from a simple story model, a user choice of topic, and a video database.

Homer was originally named after the ancient Greek story-teller. If the database provided to Homer is rich enough, and the story model is well structured, the application has a chance to live up to its name. If either one of these inputs is lacking, the application comes out looking more like Homer Simpson. If the logged database is sub-par, Homer does not have the information about what is going on in the video. If the story model is lacking, then Homer cannot apply the knowledge contained in the log to create a meaningful sequence. On occasion the application may "pull a Homer" and produce a meaningful sequence by luck. Homer relies on the story model and good logging to produce coherent narratives. In this sense Homer is really a tool for organizing the editing process. It can even be though of as a blind editing tool, or a tool in which the editor does not view the footage before using it. The choices of the editor are encoded in the story model and the information contained in the video log.

Chapter 2

Background

Homer is a video tool. Its advantages and its shortcomings are grounded in the video medium. Video is information rich. A single frame of video can take up thousands of bytes in digital form. A single image can contain a large number of identifiable objects and situations. Homer works with the same medium people may find intimidating to work with.

2.1 Video Logging

The act of logging video is an attempt to represent the variety of information contained in a video segment. Video has traditionally been logged for the editing process. Before the days of computer aided editing, video logs were used by editors to find pieces of video more quickly. A single video shoot could produce hours of video. Detailed logs allow an editor to choose promising clips of video from the hours shot without viewing it. Because of the processing power of computers, they can be used to manage the information in video logs. Homer goes a step beyond simply finding useful footage for human editors. Homer uses the information in video logs to fill a video story model. The result is a computer generated narrative. This new use of logs raises interesting questions of how video should be logged for human editors and for Homer.

At this point, text is used to log video. It is used in the form of titles, keywords and classes, and in more advanced techniques such as semantic nets and natural language models. The footage used by Homer for demonstration was logged using the Stratagraph system developed by Thomas G. Aguierre Smith [Aguierre Smith 1991]. This system allows the user to lay down descriptions over contiguous frames of video. These descriptions are in the form of class and keyword pairs. A class is a general descriptor that is meant to encompass a wide variety of related video. Keywords are specific descriptors that further describe the classes. Classes and keywords provide a limited hierarchy of description. The class provides some context for sometimes ambiguous keywords. For example the keyword "fan" could be used to mean a wind fan or a sports fan. The class of fan should clear up this ambiguity. The words "wind" and "sports" could be used as classes to provide context for the keyword "fan."

Video logging may seem like a tiresome task. It is. Other research, including the Stratification project has been working on easing the work involved in logging video. Homer does not deal with these issues; but Homer relies on good logging. The quality of a log depends on how well the information contained in the video is recorded. But since video is so information rich, the log must be selective. Based in research using ACE and Homer, it has been found that the most useful video logs contain information that is specific to the reuse of video in a story. Video must be logged to be used to create stories. The logger must keep in mind the reason that the video is being logged. Past efforts to log video have attempted to describe the content of the video and the relation between that content and its context [Eisenstein 1942]. Eisenstein dictates in his work, The Film Sense, that context is created by the juxtaposition of video images – the context of video is dependent on how it is used. This is why the eventual use of video is a key factor in logging. Its eventual use is its context. This theory allows for any number of different contexts for a single piece of video. The logger cannot hope to record all possible contexts, only the most important.

Any video that may eventually be used by Homer must be logged with Homer in mind. The important aspects of a story must be recorded. Relevant characters and places must be logged. Actions should be handled in a way in which Homer can understand their relevance and their mechanics. Often, careful choice of classes and keywords can help to describe actions. Actions can be thought of as having a period of set-up, the action itself , and an effect. The more important the action, the more attention should be paid to these aspects when logging it. The act of firing a shotgun can be logged as class:shotgun, keyword:fire. The same action can be broken up into class:shotgun keyword:load, class:shotgun, keyword:aim, class:shotgun keyword:pull_trigger, class:shotgun keyword:fire, and class:person keyword:shot. If there is a variety of video rushes of firing this shotgun, and it is going to be used as an important action in a story, then the video should be logged with at least the detail of the second example. With detailed information available, a user of Homer can model firing this gun in detail. This will help to ensure that Homer does not present a clip of the person being shot before the footage of the trigger being pulled.

2.2 Video Editing

Video editing is a thought intensive task. It is both artistic and scientific. The science of editing is in the use of the equipment, titles, and special effects. The art is in the conveying of a message or emotion through the juxtaposition of images.

An editor uses a variety of techniques, knowledge, intuition, and hard work to produce a final cut. Previous attempts at computer editors have used video logs and even simple story models to edit. Human editors use logs to organize the editing process, but they do not base their final edit solely on a log. Editors use the log to fit the video to their idea for a story. If the video does not match, the editor must either give up, or change her story. Computers rely heavily on logs to produce edits. This is a problem with computers trying to edit. Occasionally they put together a "meaningful" sequence; but computer edited pieces are sometimes better at pointing

out inadequacies in the log than at telling stories.

Homer tries to emulate the iterative process of editing. The editor must put down her story in the form of a video story model. This model is in part created using data contained in video logs. The story model is built using the keyword and class information from an existing database of logged video. Homer then applies the story to the video. Homer provides a rough edit, and a report on the video. This report is meant to give the editor some of the same information that she would glean from viewing the footage. If the story that Homer creates is not satisfactory, then the editor can use the report to revise the story model to better suit the available video.

2.2.1 A.C.E.

The Automated Content Editor (ACE) is an example of a computer editor. ACE takes keyword inputs from a user about a news topic the user wishes to know about. ACE extracts footage from its database using these keywords. ACE applies a news story model to this footage. This yields an entirely automated edit of a news story. ACE's database is a simple shot list with class/keyword descriptions. The news story model is basically a specialized filter that uses specific keywords to order the shots in an edit.

Research on computer editing using ACE highlighted some of the difficulties with such a system. A very small database was used for testing. ACE soon exhausted this. ACE even seemed to outgrow the shot bounded class/keyword descriptions. The database used by ACE was constructed quickly, and with a minimum of knowledge about how the video would be used. Certain segments were logged more appropriately than others. This became clear after only a few runs of ACE. Certain topics generated coherent news stories. For other topics, there was an obvious lack of footage or the footage was logged in little detail. This overall weakness in the database accounted for a lack of variety of coherent news stories constructed by ACE. The effect of ACE's story model on its performance is discussed in the next section.

Chapter 3

Story Models

A story model is a way of describing the various levels of structure present in a narrative. As discussed earlier, the story purpose and the actions that work to achieve that purpose are integral aspects of the story. The story model must have the capacity to represent this definitive story structure. The story model must incorporate the numerous aspects of a story. There are characters, action, plot, setting, etc. Each aspect may be more or less important than any other. The weight of each of these is specific to each story, and must be evaluated individually for each narrative.

A story model can vary in its detail of description. A detailed model of a Shakespearian tragedy could be the actual script of the play. This is a complex and specific model. On the other extreme a model could simply state "everyone dies at the end." This is an overly generalized model of a tragedy. An efficient and powerful story model obviously must lie somewhere between these two extremes. The decision of how specific to make a story model depends on how much flexibility the user wishes to give to Homer to create the story. If the description is too specific Homer may not even be able to create one full story that meets all the restrictions of the model. If the model is too general, Homer may use inappropriate footage in sections.

Computer comprehension of story models is an interesting problem. Story modeling for computers puts certain constraints on the representation. ACE used one form of

representation. Homer uses another.

3.1 ACE's Model

ACE has an internal model of what a news story should look like. For ACE every story begins with a shot of an anchorperson in the studio introducing the story. The story itself is made up of on location action shots, expert accounts, and special effect graphics. Every story closes with a shot of a reporter in the field signing off and then returns to the anchorperson in the studio for recap. This simple outline is ACE's model for a television news story.

ACE's story model was responsible for most of its success at creating meaningful narratives. It also highlighted ACE's failures. In the original version of ACE, every news story had to have an introduction, a body, and a wrap-up. When good footage was available in the database, ACE's stories were understandable. When appropriate footage was not available, the meaning of the narrative was broken. A second implementation of ACE used only footage it saw as meaningful in the video story. The narratives showed a higher rate of coherency, but the news framework broke down. If an appropriate studio shot was not available for a story, no studio shot was included. The lack of the television news style presentation detracted from the effectiveness of the presentation.

The story model for ACE was a powerful tool because television viewers are used to seeing TV news presented in a particular format. A well dressed person sitting behind a desk reading headlines says to the viewer "Get ready, here's some news." The story model of ACE mimics television news presentation style. As the second, stricter version of ACE showed, this format is important to understanding the content of the news piece. In an actual news story, a voice over is used to unify the video images that are presented. An interesting exercise is to watch television news without audio. A large body of the information is suddenly lost. In some cases the images don't even seem to form a complete story. The ACE program did not have the functionality for

voice over; but the viewer tries to piece the images and voice bytes of the body of a news piece into a story.

The viewers own perceptions of the video are an important force in story understanding. If large mistakes are not made, the viewer's desire to make sense of the video images can help in story creation. Sequences created using ACE showed that in television news, if the video presented in the body of the news story is related by a common thread, the viewer can tolerate some breadth of different images as part of one news story. The beginning and ending shots on the other hand must not conflict with the story the viewer is experiencing. For example if an anchorperson introduces a story by saying that there was a disastrous earthquake in San Francisco, any number of action shots can follow. There can be shots of fires, or of broken bridges, or even of an interrupted World Series baseball game. Yet, if the final shot of the anchorperson wrapping up the story talks about the fires, and no shots of fire were shown, the user is lost. Television news is easier to automate because of its format and the viewer's expectations. Homer must try to find similar formats for other forms of narrative.

3.2 Homer's Model

Homer's story model is built on two concepts. The first concept is a story Block; the second is a StoryLine.

3.2.1 Blocks

A story Block is a nebulous chunk of story. Even the name, Block, is ambiguous. Blocks are purposely non-nondescript. Blocks are meant to be used in a variety of ways. The way in which a Block is used determines its purpose. One of the most powerful features of Blocks is how they can be arranged in a hierarchical manner to build complex story models. Any number of Blocks can be created as subBlocks to any other Block. Every subBlock inherits all the information contained in all of its parent Blocks.(superBlocks)

The hierarchical functionality of Blocks can be used to create powerful story models quickly. The different layers in the hierarchy can be thought of as layers of story structure. Upper level blocks can be used to outline the general story purpose. Lower level blocks can be used to map specific interactions and events. Small changes to key Blocks in a hierarchical model can have dramatic effects on the story being described. The hierarchical model is both powerful and dangerous in this respect. Carefully made changes can gracefully incorporate new ideas into a story model. Careless modifications can destroy good descriptions at unexpected levels.

Each story Block contains information outlining its characteristics in the form of data fields. The most simple of these fields is the Block's name. Names perform the vital task of organizing the complex hierarchy of a story model. A consistent and powerful naming scheme allows other users to easily understand models and allows creators to reuse old models with a minimum of relearning.

The length of a block is specified in minutes and seconds. Each Block has a field that contains the desired length of the Block. A desired time is used because some edits are strictly limited by time. In television commercials a 30 second spot has to be 30 seconds. It shouldn't be less and it can't be more. Each Block also has a delta time field. The delta time specifies how much Homer can change the desired time and still satisfy the timing requirements of the Block. For example, a Block may have a desired time of one minute and thirty seconds, and a delta time of fifteen seconds. These figures give Homer the flexibility of using anywhere between one minute and fifteen seconds to one minute and 45 seconds of video to fill the Block. If the delta is set to zero, then the time field is followed strictly. Homer will fit its story to exactly the desired time if enough footage is available. If there is not enough footage that matches a story Block, then Homer will present all of the matching footage but no more.

The delta time is present to give Homer some room to work with. If the shots available for a certain block can fit well into the time plus or minus the delta, Homer

does not need to trim the shots in order to fit a specific time. The delta time allows Homer to fit the time to the story instead of fitting the story to the time.

Although the length of a block has been specified, the length in terms of shots has not been chosen. In video, the number of shots presented in a sequence and the length of those shots can be as important as the length of the sequence in time. Ten minutes of fast paced MTV style video and a ten minute continuous shot provide two totally different experiences. The differences between these experiences is due to shot pacing. Each Block has a pacing field. There are three options for pacing: slow, medium, and fast. A fast paced Block will use very short shots. A slow paced Block will use long shots; and medium paced shots will fall somewhere in between. The combination of a time plus or minus some delta and a pacing concisely and flexibly describe the desired temporal characteristics of a Block.

Every Block has two arrays of classes and two arrays of keywords. One array of classes and one array of keywords contain the entries that describe the footage that should fill a Block. These classes and keywords specify what characters appear in the Block, in what places, and performing what actions. The other two arrays contain classes and keywords that describe footage that must not appear in the Block. These arrays are used to exclude shots that may be related to the action of the shot, but do not belong in this Block. For example, there may be a story where Bill travels from New York to Boston. The introduction may want to show Bill in New York. But we definitely don't want to show Bill in Boston. To exclude the case of Bill in Boston from the first Block, Boston is added to the array of keywords that must not appear in the first Block. The ability to exclude certain shots from Blocks is a powerful tool. If you are creating a murder mystery, exclusion can be used to mask the identity of the murderer until the end. Almost all stories rely on the method of withholding information to create suspense and drama.

A key aspect of Homer's Block structure is that locks can be nested. Any subBlock inherits all the characteristics of its superBlock. This allows the author to layer

descriptions of sections of the story. A Block can describe any degree of specificity in a story. A upper level general story Block format for a conflict resolution story model may contain three large Blocks: set-up, conflict, and resolution. The lower level subBlocks can further specify actions and events.

3.2.2 StoryLines

StoryLines set up interBlock dependencies. They simply relate classes and keywords between Blocks. There are two types of relations, positive and negative. A positive relation between two classes ensures that the same keyword will be used for each class in both Blocks. A positive relation between a class and a keyword will ensure that the keyword chosen for the class in the StoryLine will match the keyword in the StoryLine. Negative relations have similar effects except that where things are the same in positive StoryLines, they are ensured to be different in negative StoryLines. A negative StoryLine connecting two classes will cause a different keyword to be chosen for each class in the two Blocks.

The idea behind StoryLines is to set up sub plots and outline the various interactions of a story. StoryLines can be used to set up important cinematic and narrative relationships. They also provide enough power for the author to destroy the story with inaccurate or misleading StoryLines.

3.2.3 Story Model Libraries

An efficient scheme for using story models is to maintain a library of general models. This library should contain models for commonly used story types such as conflict-resolution and process. The library should also contain models of specific story interactions such as conversation or travel. Combinations or parts of library story models may be used to create a single new model. This model can then be further specified using classes and keywords from a database of video specific to the current story. This method saves work in creating story models, and takes advantage of Homer's flexible general story model feature.

Chapter 4

Building a Story Model

Homer does not provide a story model building facility. Story models take the form of ordinary text files, and can be created with any standard text editor. The only important issues to be concerned with when authoring a story model is proper syntax, proper ordering, and of course the story.

4.1 Comments and White Space

Comments can only be used in a story model on a full line basis. The percent sign serves as the comment identifier. Any line that begins with a percent sign '%' is treated as a comment by Homer. The percent sign must be the first character of the line, excluding white space. A comment cannot appear on the same line as data.

White space is ignored by Homer. Any number of blank lines, spaces, and tabs can appear between lines and data objects. The only place where spaces are of a concern is in the entries of an array. This special case is discussed in the section on data arrays.

4.2 Format of a Story Block

A story Block is described by a fairly strict format. Figure 4-1 illustrates the syntax for a Block object. In the figure, all variables appear in uppercase, and time variables

```
Block BLOCK_NAME
{
 Time 00:00
 +/- 00:00

 Superblock SUPERBLOCK_NAME
 Start 00:00
 +/- 00:00

 Pacing = SPEED                                                    10

% This is a comment line

 Class
 {
  CLASSES
 }

 Keyword
 {                                                                 20
  KEYWORDS
 }


 Not_Class
 {
  CLASSES
 }

 Not_Keyword                                                       30
 {
  KEYWORDS
 }

}
```

Figure 4-1: An Example of Block Syntax

appear as "00:00." All other identifiers appear in their full form.

4.2.1 Block Name

The first variable to be entered is the BLOCK_NAME. This is the name field, which was discussed earlier. It must follow the word "Block" and be on the same line. The word "Block" signals Homer by saying "This is the beginning of a Block object." If the object being described is a subBlock of some other Block, then the word "Block" can be replaced with "subBlock." This option is included for organizational purposes. A subBlock can begin with either the "Block" or "subBlock" identifiers. The choice is purely a matter of style.

4.2.2 Open Brace

The next data item should be an open brace character '{.' Braces are used in the Block object syntax to encapsulate groups of related items. The first open brace serves to encapsulate all the data for the Block named on the previous line.

4.2.3 Block Time

The next data item in the Block syntax is Block time. As noted above, there are two times to be entered, the desired time and the delta time. The desired time follows the word "Time" and the delta time follows the characters "+/-". The "+/-" line must follow the "Time" line. All times in a Block object should be specified as they would appear on a digital stopwatch – minutes:seconds. There can only be two digits of minutes, which limits the length of any Block to be 99 minutes and 99 seconds. This should be more than enough time for any Block.

4.2.4 SubBlock Timing

The three lines that follow the Block times are only present in subBlocks. If a Block is on the highest level of the hierarchy and therefore has no superBlocks, the following three lines should not appear in the Block description. If a Block is actually a

subBlock, these lines must appear. The first of these three lines is similar to the first line of the Block, but it specifies the superBlock to the current subBlock. The syntax is the same as the first line of the Block with "SuperBlock" as the first word of the line. The second and third lines of this group specify the timing of the subBlock. The "Start" time dictates at what time in the scope of the SuperBlock does this subBlock begin. The subBlock begins at the Start time and runs for its allotted time. SubBlocks are not allowed to run over superBlock boundaries. Therefore, even if the subBlock is longer than the superBlock, the subBlock will be cropped at the end of the superBlock's time.

4.2.5 Pacing

The Block pacing is specified next. The pacing line must begin with "Pacing =." The keyword specifying the pacing of the Block follows the equal sign. The current choices for pacing are fast, medium, and slow.

4.2.6 Class & Keyword Arrays

Most of a Block object's description is taken up by four arrays of words. The four arrays are identical in syntax except for the names that begin each array. The "Class" array will be used as an example for all four. The word "Class" must appear on a line by itself. This line is followed on the next line by an open brace character '{'. This brace begins the list of words for the array. After the open brace, the words that specify the descriptive classes for the Block appear. The words must be listed one on a line. Phrases are allowed, but all spaces must be replaced by underscore characters. For example "big houses" would be replaced by "big_houses." With the exception of this replacement of spaces, the words that appear in the array must match the words used in the StrataLine database exactly. Case must be conserved and spelling must match the database exactly. A close brace character follows the last word entry. The other three arrays are entitled Keyword, Not_Class and Not_Keyword. They follow the exact same format as the Class array. The words that appear in the Class and

```
StoryLine
{

  % Comments and white space are treated exacIty
  % as they are treated in Block objects.

  FromBlock BLOCK_NAME
  ToBlock BLOCK_NAME

  Class/Keyword WORD                                          10
  Class/Keyword WORD

  Negative/Positive

}
```

Figure 4-2: An Example of StoryLine Syntax

Keyword arrays are classes and keywords that describe video that should appear in the current Block. The entries in the Not_Class and Not_Keyword arrays are classes and keywords that describe what should not appear in the current Block.

4.2.7 Close Brace

The final line of the Block object is a close brace character '}'. This close brace matches the open brace that appeared on the second line of the Block object.

4.3 Format of a StoryLine

The primary purpose of StoryLines is to set up dependencies between Blocks. These dependencies can be related to the action of a story. They can describe what characters or objects can and cannot appear. The dependencies can also set up cinematic constraints. StoryLines can be used to prevent cinematic taboos such as jump cuts and 180 degree cuts.

4.3.1 The First Two Lines

The syntax for a generic StoryLine is shown in figure 4-2. The first line contains only the word "StoryLine." This line tells Homer that the following lines comprise a StoryLine object. The line that follows this contains an open brace character '{'. The information following this character is the data of the StoryLine object.

4.3.2 To and From Blocks

A StoryLine can be pictured as a directional link, or an arrow connecting two Blocks. The arrow must originate from one block and point to another. The Block from which the arrow originates is the "FromBlock." The first data line after the open brace begins with the identifier "FromBlock." The Block name that follows refers to the first Block of the StoryLine. Similarly, the next line begins with the identifier "ToBlock." This is the destination Block of the arrow representing the StoryLine. The Block name that follows is the second Block of the StoryLine object.

4.3.3 Classes and Keywords

The next two lines of data contain the classes and keywords that are related by the StoryLine. The first line describes the class or keyword of the FromBlock. The first word of this data line is either "Class" or "Keyword" depending on which is to be used. If the author wants to link a class in the FromBlock with the StoryLine, then "Class" should appear as the first word. The second word is the class itself. Similarly if "Keyword" is chosen as the first entry, the second entry must be an actual keyword from the FromBlock's descriptive arrays. The second line is identical to the first except that it describes the word to be linked in the ToBlock.

4.3.4 StoryLine Temperament

The last data line of the StoryLine describes its temperament. The two choices for this line are "Positive" and "Negative." The temperament of the StoryLine, along

with whether classes or keywords are chosen for each Block, describe the type of relation that is set up by the StoryLine.

If a positive temperament is chosen, the StoryLine will enforce a equality relationship between the FromBlock and the ToBlock. If a class is chosen for each Block, the keywords chosen to describe these classes will be the same in both Blocks. If a class is chosen for one Block and a keyword is chosen for another, a different relationship is set up. In this case, the keyword chosen for the class will be the keyword that is specified for the other Block in the StoryLine.

A negative temperament will create relationships opposite to the relationships set up by a positive temperament. If two classes are used, the keyword chosen for these two classes will be different. If a class and a keyword are chosen, the keyword specified will not be the one used to describe the class.

All these relationships are enforced in the segment generation procedure of the application. This means that different relationships can be specified within the language of a StoryLine object. Any user that can program her own segment creation routine can also create new or modified StoryLine relations. The current segment creation procedure of Homer follows the relations described in the previous two paragraphs. However, these relations are only taken as indicators for Homer. In other words, Homer will enforce the relations specified by StoryLines as long as they do not interfere too seriously with other restrictions active in the story model.

Chapter 5

Story Reports

A Story Report is a collection of information that is used to model the iterative act of video editing. A Story Report basically tells the user how well a story model matches a video database. The current version of a story model provides a simple yet useful summary of this information.

5.1 Report Format

Figure 5-1 shows the format of a blank story Report. This format is automatically generated by Homer when a story model is applied to a video database. The Report is broken down on a Block level. It gives the total time of footage available, and the number of segments that make up that total time. The requested time of the Block is also shown to put the amount of footage available in perspective.

Report for BLOCK_NAME

Total time of footage == 00:00

Time of Block = 00:00
+/− 00:00

Number of segments of footage = 0

Figure 5-1: An Example of Report Format

The footage included in this summary consists of any video that has keywords or classes in their description that match the entries of the class and keyword arrays for the Block in question. In other words this is a collection of loosely associated video. A stricter matching scheme can be implemented, but this would involve a stricter sequence generation routine.

5.2 Use of the Story Report

The information presented in a Report is an attempt to give the user an indication of how their story model is being interpreted by Homer. A Report only gives a general view of Homer's perspective. This basic information can be used to build more effective story models quickly.

In order to best utilize the story Report, it should be reviewed with the story Block descriptions. The story model author can quickly see what Blocks need adjusting. If a specific Block calls for two minutes of video, and the Report shows only one minute available, something has to give. There is simply not enough video in the database to match the description of the story model. The length of the Block has to be shortened or the Block's description has to be loosened. Manipulations of Block times are straightforward. If the user wants to relax the description of a story they can look to eliminate entries in the Not_Class and Not_Keyword arrays of the Block description. If this does not produce the desired affect, then more keywords or classes can be added to the descriptive arrays.

The opposite problem can also occur – a Block's description can be too general. In this case there may be many more minutes of footage available than are necessary. This may or may not be a problem. If the actual video story produced by Homer is satisfactory, then there really is no problem with general Block descriptions. This may often be the case because Homer orders the matching video by how well the description of the footage matches the Block description. The best matched video is tried first, and then increasingly less related footage is used to fill the story Block.

In some cases, the footage chosen by Homer for a Block may not match the

expectations of the author. This is often the result of a lack of context when choosing classes and keywords. For example, a user doing a story on the San Francisco area may choose the class:keyword pair "place:marina" in a story Block. The user may envision a place where boats are docked in the San Francisco bay. Homer may present the user with images of a mid to upper middle class neighborhood. The application is simply matching the class:keyword pair to the descriptions of the video. It has no real understanding of the meaning of the word marina; but the person who logged the video had a definite understanding. In this example the logger was describing a neighborhood in San Francisco known as the Marina section. The lack of context in descriptions is a problem inherent to describing video. Homer does not try to solve this problem. Instead, Homer offers an iterative approach to circumventing it. A story model is not a static entity. It is meant to be changed to fit the needs and desires of each individual user. The story Report is one tool to help in this process. The video story produced by Homer is another. Therefore, in our example, if the user wants to see boats, then he must return to his story model, and try to adjust it to the style of the database in use. There is also a second option – use a different database of video.

Chapter 6

Object Oriented Programming in Homer

Homer was implemented in the C++ object oriented programming language. The use of C++ provides substantial gains over the use of C. These gains include easier updates, reusable code among applications, and abstraction from performance.

The first task in implementing Homer was to create an object set for dealing with story models and StrataLine log entries. A number of data objects were created, including Blocks, StoryLines, Reports, and StrataLines. The specifications for these objects are listed in appendix D. The code for each object is listed in appendix E. These objects were made to handle the types of data specific to the application.

6.1 Blocks

The Block object was created to hold all the information contained in a single story Block. This includes timing, pacing, and class/keyword arrays. The Block object also holds all the StoryLines that are related to the Block. An array of StrataLines that is contained in the Block stores copies of all the video that matches the Block's descriptions. The Block is the central data object of Homer.

The other factor that is incorporated into Blocks is the potential use in a hier-

archical structure. All Blocks contain links to potential superBlocks in the form of superBlock names. These subBlocks also hold the time at which they affect their superBlock's description. A second C++ object, called a BlockArray, is a mutable array of Blocks. This object is used to hold the hierarchy of Blocks in a story model.

6.2 StoryLines

A StoryLine object is basically a link between two Blocks. The StoryLine holds the names of the two Blocks being linked, and the characteristics of that link. StoryLines in a story model are eventually stored in arrays in the two Blocks that the StoryLine links. A mutable array of StoryLines, called a Plot, is used to hold a list of StoryLines.

The names StoryLine and Plot were chosen to describe their function. StoryLines should be used to set up various dependencies between characters and objects in a story. The combination of these StoryLines should outline the Plot of a narrative.

6.3 StrataLines

A StrataLine is a piece of video description. It links a piece of video from an in frame and out frame to a group of descriptive classes and keywords. These were designed as a part of Thomas G. Aguierre Smith's Stratification system, and are used by Homer. A file containing a group of StrataLines is used by Homer as a logged video database file. A StrataFile data object is used to store a group of StrataLines.

6.4 Reports

The ReportArray object is used to store information about how well a story model matches a database. The total ReportArray is made up of any number of individual Reports. The Report object is general. The name field of a Report can refer to any data item that can be described with a name. The other fields of a Report keep track of how much video matches the named item. This object has been designed to

meet various levels of description. Therefore, when a Report object is unparsed, or outputed, some context information is necessary to complete the description of the Report. This contextual information may include the type of data item to which the name field refers and the requirements for video to match this particular item.

6.5 Object Set Benefits

The use of the data object set allows the issue of performance to be ignored. Originally the object set's memory management was implemented in a grossly inefficient manner. Because of the level of abstraction supplied by the object set I was able to write and test procedures that depended on memory management. I was able to rewrite the data management routines in the object set at a later date. After this change was made, one compilation was all that was needed to see the dramatic improvement in performance. Not one line in the upper level procedures that relied on the data objects had to be changed. Future updates to the object set can be implemented in the same fashion.

Chapter 7

Homer, the program

The complete source code for Homer is listed in appendix F. The following is a description of the processes carried out by the source code.

7.1 The Story Model File

Homer's first action is to read in the story model file containing all Blocks and StoryLines for the current model. The Blocks of the story model are stored in a BlockArray object. The StoryLines in the model are stored in their affecting Blocks. Then the hierarchy is set up. All subBlocks inherit the classes, keywords, and StoryLines of their superBlocks.

7.2 The Video Database File

Next Homer parses the video database file into a StrataFile object. The database file contains StrataLines describing the footage available to Homer. This file is compatible with Thomas G. Aguierre Smith's Stratification project [Aguierre Smith 1991] and Hiroshi Ikeda and Hiroaki Komatsu's Infocon video incon system. Once the StrataLines are read in, they are broken down into their finest granularity of description. The Stratification logging system allows the logger to describe footage in a concise form. Homer must pull all of the descriptive information from this abbrevi-

ated form. Homer marks all the points in the footage where description changes and creates a new StrataLine for each uniquely described section of video.

7.3 Story Model Meets Database

Once the story model and the database have undergone this preliminary processing, the database is applied to the story model. In this procedure the StrataLines that match the descriptions for each Block are simply copied into the Block_Strata object for each Block. Now the BlockArray contains all the Block, StoryLine, and StrataLine information needed to create a story.

The actions performed thus far are the main functions of Homer. From this point on any number of report generation procedures and sequence creation routines can be applied. All the basic information contained in the story model and video database has been loaded in and set up. The following sections describe the functions that organize the output.

7.4 Report Generation

Any number of report generators can be applied to Homer. In the current implementation, the Report contains information on the Block level. The report generator calculates the amount of available footage for each Block and presents this information as a total time and a number of shots. Future implementations may report database matching on the class and keyword level for each Block.

7.5 The Sequence Creator

The sequence creator decides what pieces of video will be used in the story. The information it has available to make these decisions are the the video for each Block, the story Block descriptions, and the StoryLines. Different procedures may use each of these in different capacities.

In order to make the sequence generator's job easier, one more procedure is applied to the BlockArray. In this procedure, the hierarchy of Blocks and subBlocks is flattened. This is similar to the operation that is performed on the StrataLines to extract their full description. The finest granularity of description by story Blocks is found. No information is lost in the flattening process. This procedure is done to simplify the management of time in the story building process. Once the hierarchy is flattened, the sequence generator only needs to iterate through the array of Blocks, filling each with video.

Now all the information that can be used to create a story is available in a simple form. The procedure that chooses footage to present for the story only needs to read one Block at a time and choose StrataLines from its Block_Strata list. But there are many ways that these StrataLines can be chosen. A strict methodology can be used, and only StrataLines that fully match the description of a Block will be chosen. A loose methodology, where any StrataLine that matches can be chosen, is also a possibility. One sequence generator may demand the enforcement of StoryLine relationships, while using Block descriptions only on an advisory level. Another sequence generator may make all of its decisions based on the Block descriptions and use the StoryLines only to break ties. Generally there will be some algorithm by which the best shots are chosen for each Block. But some users may want to use different values to determine which shot is best. That is why all the processing of the BlockArray and database is done first. After this initial processing, any number of procedures can be applied to choose the shots to be used in a story. A user with some basic programming experience can even code his own shot selection procedure.

The current sequence generator uses all information available in a relatively equal way. The factors that it takes into account are Block arrays of classes and keywords, Block pacing, and StoryLines. Each StrataLine that matches at least one class or keyword from the Block arrays is evaluated. A weighting function is used to determine which StrataLines of video should be used for each Block. Every match to a class or keyword increments the weight. Therefore if a StrataLine matched two classes and one

keyword, its weight will increase by three. If the pacing of a StrataLine matches the Block's pacing value, the StrataLine's weight is increased by one. Therefore pacing has a slightly lesser effect than keywords and classes. StoryLines affect pacing in a different manner. The weight of a StrataLine is increased for each StoryLine that it satisfies. The StrataLine's weight is incremented by the number of StrataLines in the other Block that is pointed to by the StoryLine that satisfy the StoryLine. In other words, the StoryLine points to some Block other than the current one. Any number of StrataLines in that Block may satisfy the current StoryLine. It is that number of StrataLines that is added to the weight of the StrataLine in the current Block. The weighting scheme puts at least as much emphasis on StoryLines as it does on classes and keywords. Each StrataLine for a Block is weighed. The StrataLines with the highest weights are used to fill the Block.

Chapter 8

A Trial Run

Homer was tested using fifteen minutes of video taken from a larger body of footage shot by Ricky Leacock. The subject of the video is Joey Arias, an Andy Warhol impersonator. The fifteen minutes of video show the ritual that Joey undertakes to become Andy Warhol.

This video was used to model process. The overall subject, becoming Andy, is a process. This larger goal is made up of smaller processes such as applying makeup and getting into the right mental state.

8.1 Logging the Video

The first order of business was to log the video of Joey. A small list of classes and keywords were used. The classes used were; Actions, People, Things, Monologues, and Framing. Framing is a general class that is used to describe how tightly the shot was framed. The remaining four classes were designed specifically for this video. The listings of the keywords for the various classes are contained in appendix A. A total of 102 StrataLines were created for the 15 minutes of video. The list of these descriptions is contained in appendix B.

The VIXen video logging system, developed by Joshua Holden was used to create StrataLines that describe the video. One logging pass was made over the entire video for each class. In each pass all the appropriate keywords for that class were assigned

to the video that they described. For example, during the pass for the Things class, all the shots in which Joey was either wearing or trying on wigs were assigned the class:keyword pair Things:wig. Each pass was completed in less than the total time of the video. This speed was achieved by scanning quickly though sections and only marking the relevant in and out points of descriptions. The time for each pass varied depending on how strictly the class related to the audio. The class Monologues was the slowest to log. This was because the audio had to be followed carefully. Because of this, the video had to be viewed at regular play speed for most sections in order to follow the audio track.

Several other factors influenced the style of the video log. One such factor was the style of the video itself. The majority of the video is comprised of Joey speaking to the camera while making himself up to be Andy. The actions in the video are small and involve mostly motion contained in a small space. All the video is taken from one angle. This removes the responsibility of creating a space in the story model. The space is constant. The framing in the video varies from medium shots to extreme close-ups. The framing of each shot was logged to be used to avoid jump cuts.

A major factor that influenced the log was audio. Most of the audio in the footage is comprised of short comments or sound bytes. All StrataLines were logged on audio boundaries. In other words, StrataLines always started on a pause and ended on a pause. If StrataLines began and ended in the middle of sentences or words, Homer would break the video up into these small segments. Then the story that Homer produced would be full of half sentences and cuts in the middle of words. Because the audio cannot easily be separated from the video, logs for Homer must be constrained by the audio track.

8.2 Building the Story Model

Once the video has been logged, the story model building begins. An incremental story building process was used. In other words, a complex story model was built from a simple model by adding levels of description. This method was good for

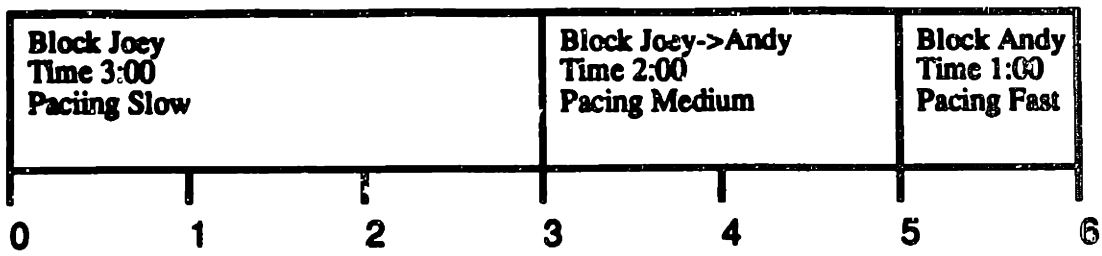

Figure 8-1: An illustration of the Primary story model

establishing continuity and creating transitions.

8.2.1 The Primary Model

The primary story model was comprised of three Blocks. These three Blocks mapped Joey Arias's transition from Joey to Andy Warhol. The primary story model is listed in appendix C. An illustration of the Block structure for the primary model is shown in figure 8-1. The first Block has one descriptive keyword – Joey_Arias and one exclusion keyword – Andy_Warhol. These keywords constrain the first Block to contain shots of Joey and not Andy. The video used in the first Block should show Joey being himself. The second Block contains both Joey_Arias and Andy_Warhol as descriptive keywords. The Block will look for footage where it is unclear whether Joey is being himself or acting like Andy. Some of this footage may be of Joey turning into Andy. The final Block has Andy_Warhol as a descriptive keyword and Joey_Arias as an exclusion keyword. This Block will contain footage of Joey acting like Andy Warhol.

The timing of the story was set to run from slow to fast. The first Block was three minutes long and had slow pacing. The second Block was two minutes long with medium pacing. The third Block was only a minute and was fast paced. The thought behind this timing was to create a temporal buildup to a climax.

The video produced by the initial story model was partially successful. The overall transition from Joey to Andy was present. The increased pacing of the video was

also effective. But there were major problems with continuity. The most obvious of these involved Joey's blonde wig. There were shots of Joey with and without his wig on interspersed throughout the first two Blocks of video. This broke the feeling of the continuous process of Joey turning into Andy. This problem and several other breaks in continuity were easily fixed by adding levels of description to the initial story model.

To solve the wig continuity problem, a level of subBlocks was added to the Joey Block. This level had only two subBlocks. The first subBlock affected the first half of the Joey Block and the second subBlock affected the second half of the Joey Block. The first subBlock listed wig as an exclusion keyword. This ensured that Joey would not be wearing the wig for the first half of the first Block. The second subBlock listed wig as a descriptive keyword. This told Homer to show Joey in his wig for the second half of the Block. The video output of this updated model successfully maintained continuity with respect to the wig. Once Joey put his wig on, it was on for good.

However, the Joey and Andy, and the Andy Blocks did not have any keywords that specified the wig. In this example, all the footage chosen for these two Blocks, showed Joey wearing his wig. If this were not the case, a subBlock could be added for each of these Blocks specifying that the wig should be present.

8.2.2 The Complex Model

This process of adding constraints to improve the quality of the story, and viewing the product was carried out several times. Eventually a thickly described coherent model was developed. The text of the model is listed in appendix C. An illustration of the Block structure for the complex model is shown in figure 8-2. The most surprising quality of this process of improving the story model was that it was fun. Making changes to the story model and seeing them affecting the video edit provided instant satisfaction. The experience truly became one of building a story.

The process of refining the original story model highlighted certain problems. For the first several iterations the affects were successful. Soon however, the limits of the database were reached. The story model became specific enough to call for shots

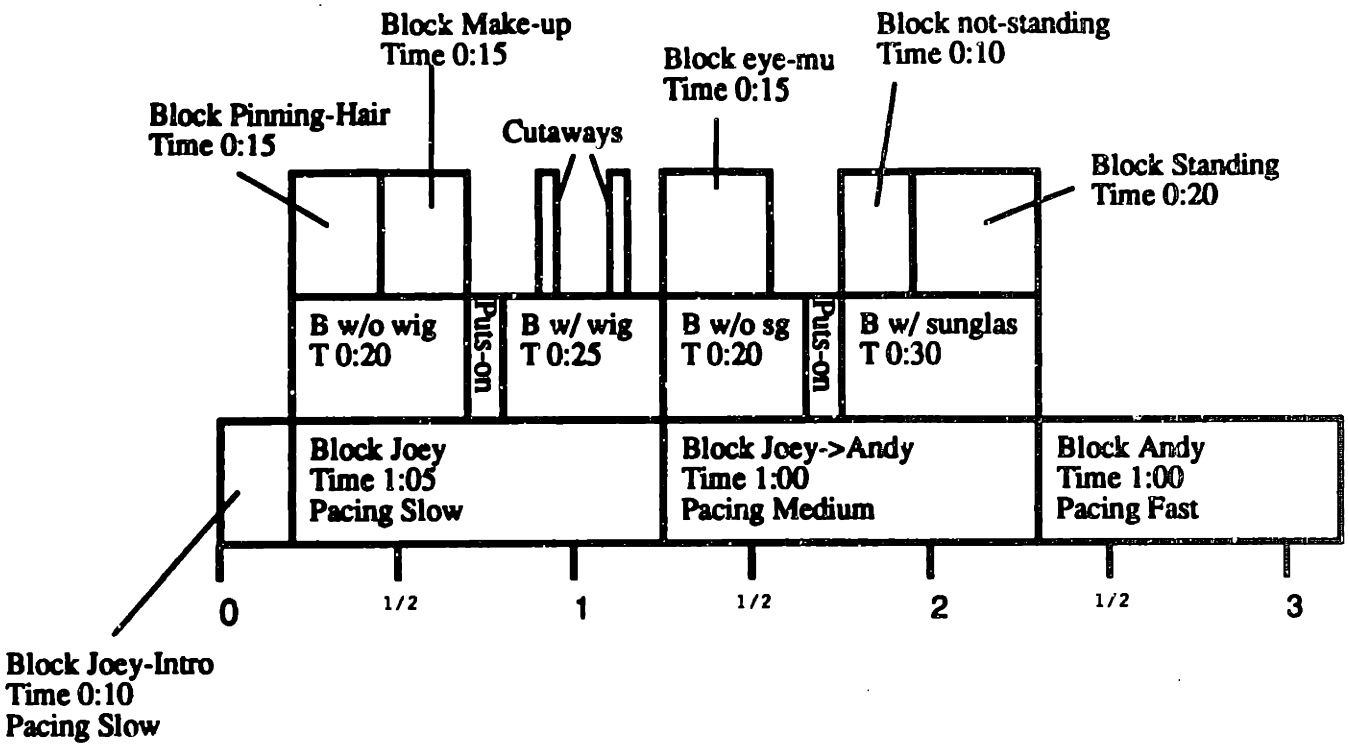

Figure 8-2: An illustration of the Complex story model

that either were not present in large enough quantity or not present at all. In even worse cases, shots were chosen for particular purposes, but their expected content did not match their description. These occurrences were rare, but that may be because I logged the footage and built the story models. Footage logged by a user other than the story model author should encounter the problem of description/content conflicts more often. This is due to the different contexts of the logger and story model author. In the case where the same person logs and authors, there can still be a contextual difference based on how the person is thinking at the different stages of story development.

Although the complex model suffered some problems, the video produced was significantly more comprehensible than the primary model's result. Also, many of the slight breaks in continuity in the complex model could be fixed by reordering a small set of the shots in the video example.

As a user develops a story model, she asserts more control over the edited result. As subBlocks become smaller, and the layering of descriptions becomes thicker, the user approaches specifying images at the shot level. Once the shot level is reached,

the story model author has become an editor.

At the shot level of control the timing of various blocks becomes important. A single block can be filled by any number of shots. This depends on the length of the Block and the length of the shots. When the author begins to reach the shot level, the times of the subBlocks involved become restricted so that one shot will fill the Block. This level was encountered in the complex story model. This level of control allowed a cutaway to be added to the story. In an early model, a shot showed up of Joey talking to a person off screen other than the cameraman. A subBlock was added after this shot that specified an image of a person other than Joey. This person was Sarah. The subBlock also had the restriction that Joey should not appear in the shot. The effect of this change was that Joey was shown talking off camera, and the following shot showed Sarah listening.

Chapter 9

Conclusion

The development and use of Homer have provided insights into the areas of video logging, editing, and story generation. Ideas about story models and story structure have been explored; and directions for continued research have been marked.

9.1 Story Models and Story Structure

The trial run showed that Homer can be used to successfully model process. This was done by building larger processes from small processes and parts of processes. In the example, the smaller processes of applying makeup, putting on a wig, and putting on sunglasses were combined to create the larger process of becoming Andy Warhol.

The second major story type, conflict resolution, was not tested. Future research will concentrate on building conflict resolution story models using Homer. One possible approach is to build the portions of the conflict resolution model from processes. Research using Homer should be able to determine how well this approach works.

Story models were even able to address some primitives of editing. The shot pacing was specified, and was followed closely. Editing techniques such as cutaways can be implemented using Homer.

The one feature that was not tested in the trial run was StoryLines. The hierarchical Block structure stole the spotlight in the story modeling process. However, a potential use of StoryLines did emerge from the example. On several occasions,

Homer produced a jump cut in the final edit. This is when two shots of the same subject with the same framing follow each other. Once the shot level is reached by the Block structure, StoryLines could be used to avoid this cinematic error. A simple chain of negative StoryLines relating the Framing class would eliminate jump cuts. Each StoryLine ensures that two adjacent shots do not exhibit the same framing. If one such StoryLine connects every pair of connected shot level subBlocks, jump cuts would be completely eliminated from the story.

9.2 Logging Issues

Advanced use of Homer led to some interesting discoveries about the video log. At some point, a detailed story model reaches the limits of the log. Certain Blocks in the story model may request video that is present, but has not been logged in the way it is requested. The story model may also become more detailed than the video log. This case appeared in the complex story model in the trial run.

A section of the complex story model described Joey trying on sunglasses. The Block was short and used Joey and trying_on_sunglasses as keywords. Two shots were chosen by Homer to fill this Block. The first shot showed Joey taking off one pair of sunglasses and putting on another. The second shot showed Joey with no sunglasses on, picking up a pair of sunglasses in a case. Homer presented these two shots in the wrong order, and the story's continuity was broken. The story model could be enhanced by adding two subBlocks to the trying on sunglasses Block. The first subBlock would specify picking up the sunglasses and the second subBlock would specify putting on the sunglasses. However, this would not have affected the video edit produced by Homer, because the action of trying on the sunglasses is not logged to this level of specificity.

The story specified in the previously described story model asks for a certain level of specificity, but the log cannot deliver it. A simple solution is to have the user switch the order of the two shots of putting on sunglasses. This fixes the problem in the edit, but it does not solve the problem for Homer. If the same story model is run

through Homer with the same database, Homer will still place the two shots in the wrong order. The solution is to have Homer do some work to help himself.

Homer has all the information he needs to solve the problem. The story model specifies that Joey should pick up the sunglasses in one Block, then put them on in the next. The video story produced by Homer reflects the story model, but it has certain problems. It is simple for the user to see these problems, and in some cases it is equally easy for the user to fix them. Once the user has changed the ordering in the edit, Homer can incorporate this knowledge into the log. Homer just has to run in reverse.

9.3 Homer in Reverse

Homer takes story models and applies them to logs to produce video stories. In reverse, Homer can take story models, apply them to video and produce logs. Homer can add entries to the log based on the information in the story model and the users improvements to the edit. In the example of the sunglasses, Homer can add the descriptions of picking_up_sunglasses and putting_on_sunglasses to the proper video segments. When Homer recreates an edit for this story model, it should get the trying on sunglasses sequence right. This is essentially a form of learning. Homer records the information in the story model and the user's modifications to the edit in the log. Homer learns from this procedure, and does not make the same mistake again.

Running Homer in reverse can be viewed as an alternative to logging. All the work involved in video logging is still present, but the focus has changed. In the act of conventional video logging, the logger attempts to add a large number of layers of description to video one description at a time. By using a story model, the user can incorporate the same mass of information into a more intuitive form.

When performing conventional logging, the logger is constantly asking himself about what the video contains. If the logger is new to the video, this process is necessary. If the user is familiar with the video, he may already know how the video should be used and therefore logged. Viewing the video while logging may

end up being distracting to the logger. Logging is a labor intensive task. Logging while the video is running is difficult because the logger must maintain concentration throughout the entire procedure. If the logger losses concentration he may have to backtrack. In worse cases, the logger may lose his concentration, and thereby the context in which he is logging. This may result in discontinuities or even errors in the log. Because the video and the log are passing in time, it is easy for the logger to allow these errors to pass. Some of these issues can be resolved by using story models for logging.

A story model can be viewed as a log for a story. A complete log and a complete story model contain essentially the same information. The main difference between a story model and a log is that the story model is not as grounded in the video. The story model offers a layer of abstraction from the video. This allows the author to concentrate on the issues involved in the story without being a slave to the video. Story models could even be used to apply description directly to video. When someone goes out to shoot or returns from shooting they generally have some form of story model in mind. If the interface for building story models is good, it should be a straightforward process to create this story model. Then, the major in points and out points of the footage can be inserted into the model; and the video has been logged. The same process can be done when creating an edit manually. Every editor has a story model for their edit. If they build the story model to match their edit, the footage used in the cut will be automatically logged to the precision of the story model.

These alternate forms of logging seem promising. Future research will be needed to discover just how effective they will be. One fact that supports this research is how the two tasks of logging and story building differ. Logging is a chore, but story building can be fun.

Appendix A

Classes and Keywords for Trial Run

A.1 Things

- white make-up
- eye pencil
- wig
- sunglasses
- hair brush
- hair pins
- mirror
- dog
- hair spray
- black turtleneck

A.2 People

- Joey Arias
- Andy Warhol
- Sarah
- Ricki Leacock

A.3 Actions

- greeting
- brushing hair
- pinning hair
- applying make-up
- trying on wigs
- putting on wig
- watching
- brushing wig
- spraying wig
- applying eye pencil
- trying on sunglasses
- making-up hands
- standing
- showing hand positions

- waving
- uncovering

A.4 Monologues

- become Andy in your neighborhood
- I'm going to become Andy
- chaio Manhattan
- how Andy got pale
- how to make make-up
- fast talking
- falling into Andy mode
- Andy's age
- Andy's interview on cable
- Andy's real hair
- Andy's new wig
- Andy's different wigs
- Andy without wig
- Andy's snap on wig
- how to become Andy
- what to think
- questions

- slipping into Joey
- finding out about a person
- father of pop
- I love make-up
- I love black
- the Andy dance
- your fifteen minutes

A.5 Framing

- Extreme Close-up
- Medium Close-up
- Full Close-up
- Wide Close-up
- Close Shot
- Medium Close Shot
- Medium Shot
- Medium Full Shot
- Full Shot

Appendix B

StrataLines in Video Log for Trial Run

The format is as follows

- VolumeName|InPoint|OutPoint|ContentFrame|Speed|Title(none)|Class|Keyword

1. Warhol|719|8687|24467|30| |People|Joey_Arias
2. Warhol|8716|9309|754|30| |People|Sarah
3. Warhol|9309|13477|1088|30| |People|Joey_Arias
4. Warhol|11410|24201|1767|30| |People|Andy_Warhol
5. Warhol|14302|16696|3011|30| |People|Joey_Arias
6. Warhol|17723|17965|4738|30| |People|Joey_Arias
7. Warhol|19941|22115|4257|30| |People|Joey_Arias
8. Warhol|24201|24734|6851|30| |People|Joey_Arias
9. Warhol|719|789|7117|30| |Things|hair_brush
10. Warhol|934|1243|8534|30| |Things|wig

11. Warhol|1518|2016|9310|30| |Things|hair_brush

12. Warhol|2655|3368|9310|30| |Things|hair_pins

13. Warhol|3369|6108|9666|30| |Things|white_make-up

14. Warhol|4195|4319|9666|30| |Things|dog

15. Warhol|6851|7116|11334|30| |Things|white_make-up

16. Warhol|7117|8534|16056|30| |Things|wig

17. Warhol|8534|8687|18266|30| |Things|hair_pins

18. Warhol|9310|9666|23463|30| |Things|hair_brush

19. Warhol|9310|24734|23503|30| |Things|wig

20. Warhol|9666|11066|724|30| |Things|hair_spray

21. Warhol|11334|14895|855|30| |Things|white_make-up

22. Warhol|14895|16056|1498|30| |Things|eye_pencil

23. Warhol|16056|24515|2210|30| |Things|sunglasses

24. Warhol|18266|20375|2529|30| |Things|white_make-up

25. Warhol|23463|23544|2876|30| |Things|white_make-up

26. Warhol|719|729|3233|30| |Framing|Extreme_Close-up

27. Warhol|729|981|3233|30| |Framing|Close_Shot

28. Warhol|981|2016|3755|30| |Framing|Medium_Close_Shot

29. Warhol|2016|2405|5715|30| |Framing|Close_Shot

30. Warhol|2405|2654|7607|30| |Framing|Medium_Close_Shot

31. Warhol|2655|3098|8305|30| |Framing|Medium_Close_Shot

32. Warhol|3098|3368|8925|30| |Framing|Close_Shot

33. Warhol|3369|4141|9235|30| |Framing|Medium_Shot

34. Warhol|4141|7290|9489|30| |Framing|Close_Shot

35. Warhol|7290|7924|9895|30| |Framing|Medium_Shot

36. Warhol|7924|8687|12536|30| |Framing|Close_Shot

37. Warhol|8688|9162|15227|30| |Framing|Full_Close-up

38. Warhol|9162|9309|16031|30| |Framing|Full_Close-up

39. Warhol|9310|9668|18115|30| |Framing|Close_Shot

40. Warhol|9668|10123|19715|30| |Framing|Medium_Close_Shot

41. Warhol|10123|14950|21851|30| |Framing|Close_Shot

42. Warhol|14951|15504|22578|30| |Framing|Extreme_Close-up

43. Warhol|15504|17965|22734|30| |Framing|Close_Shot

44. Warhol|17966|18265|22898|30| |Framing|Medium_Close_Shot

45. Warhol|18266|21165|23024|30| |Framing|Medium_Shot

46. Warhol|21165|22538|23620|30| |Framing|Close_Shot

47. Warhol|22539|22618|24438|30| |Framing|Full_Close-up

48. Warhol|22619|22849|24438|30| |Framing|Extreme_Close-up

49. Warhol|22850|22947|1106|30| |Framing|Medium_Close-up

50. Warhol|22947|23101|1755|30| |Framing|Full_Close-up

51. Warhol|23101|24139|2336|30| |Framing|Wide_Close-up

52. Warhol|24140|24737|3011|30| |Framing|Medium_Shot

53. Warhol|719|1493|4734|30| |Actions|greeting

54. Warhol|1493|2018|6434|30| |Actions|brushing_hair

55. Warhol|2018|2654|6943|30| |Actions|separating_hair

56. Warhol|2655|3368|7824|30| |Actions|pinning_hair

57. Warhol|3369|6099|8609|30| |Actions|applying_make-up

58. Warhol|6099|6770|8794|30| |Actions|smoothing_make-up(hands)

59. Warhol|6770|7116|9259|30| |Actions|applying_make-up

60. Warhol|7117|8532|9408|30| |Actions|trying_on_wigs

61. Warhol|8532|8687|10364|30| |Actions|pinning_hair

62. Warhol|8717|8871|13117|30| |Actions|watching

63. Warhol|9210|9309|15476|30| |Actions|watching

64. Warhol|9310|9678|17010|30| |Actions|brushing_wig

65. Warhol|9678|11050|19225|30| |Actions|spraying_wig

66. Warhol|11337|14897|20867|30| |Actions|applying_make-up

67. Warhol|14897|16055|21751|30| |Actions|applying_eye_pencil

68. Warhol|16055|17965|21751|30| |Actions|trying_on_sunglasses

69. Warhol|18103|20348|22327|30| |Actions|applying_make-up_to_hands

70. Warhol|20348|21387|22578|30| |Actions|standing

71. Warhol|21387|22115|24261|30| |Actions|showing_hand_positions

72. Warhol|22116|22538|24673|30| |Actions|greeting|Actions|waving

73. Warhol|22539|22618|24673|30| |Actions|goodbye|Actions|waving

74. Warhol|24140|24382|24673|30| |Actions|uncovering

75. Warhol|24611|24735|1107|30| |Actions|goodbye

76. Warhol|719|1496|3237|30| |Monologues|I'm_going_to_become_Andy

77. Warhol|3106|3368|3679|30| |Monologues|chaio_Manhattan

78. Warhol|3505|3853|4030|30| |Monologues|how_Andy_got_pale

79. Warhol|3853|4208|4030|30| |Monologues|how_to_make_make-up

80. Warhol|4972|5379|5175|30| |Monologues|fast_talking

81. Warhol|5379|5782|5580|30| |Monologues|falling_into_Andy_mode

82. Warhol|5782|5993|5887|30| |Monologues|Andy's_age

83. Warhol|6532|6730|6631|30| |Monologues|Andy's_age

84. Warhol|7545|7758|7651|30| |Monologues|Andy's_interview_on_cable

85. Warhol|7863|8273|8068|30| |Monologues|Andy's_real_hair

86. Warhol|9519|9669|9594|30| |Monologues|Andy's_new_wig

87. Warhol|9833|10023|9928|30| |Monologues|Andy's_new_wig

88. Warhol|10116|10596|10356|30| |Monologues|Andy's_different_wigs

89. Warhol|11073|11339|11206|30| |Monologues|Andy's_snap_on_wig

90. Warhol|11593|13087|12340|30| |Monologues|how_to_become_Andy

91. Warhol|13087|13601|13344|30| |Monologues|finding_out_about_a_person

92. Warhol|13675|14285|13980|30| |Monologues|questions

93. Warhol|14285|14431|14358|30| |Monologues|slipping_into_Joey

94. Warhol|14431|14766|14598|30| |Monologues|finding_out_about_a_person|Monologues|questions

95. Warhol|18266|18876|18571|30| |Monologues|father_of_pop

96. Warhol|19063|19314|19188|30| |Monologues|I_love_make-up

97. Warhol|19314|19924|19619|30| |Monologues|I_love_black

98. Warhol|19924|20348|20136|30| |Monologues|how_to_become_Andy

99. Warhol|20451|20890|20670|30| |Monologues|how_to_become_Andy|Monologues|the_Andy_dan

100. Warhol|21158|21859|21508|30| |Monologues|how_to_become_Andy|Monologues|the_Andy_dan

101. Warhol|22619|22849|22734|30| |Monologues|your_fifteen_minutes

102. Warhol|22850|24139|23494|30| |Monologues|become_Andy_in_your_neighborhood

Appendix C

Story Model Examples

C.1 Primary Story Model

```
%__Basic story model of Joey becoming Andy

Block Joey
{
 Time 03:00
 +/- 00:30

 Pacing = Slow

 Class                                                    10
 {
 }
 Keyword
 {
  Joey_Arias
 }

 Not_Class
 {
 }                                                        20
 Not_Keyword
```

```
 {
   Andy_Warhol
 }
}
%-------------------------------------------------------------
Block Joey->Andy
{
 Time 02:00
 +/- 00:20                                                    30

 Pacing = Medium

 Class
 {
 }
 Keyword
 {
   Joey_Arias
   Andy_Warhol                                                40
 }

 Not_Class
 {
 }
 Not_Keyword
 {
 }
}
%-------------------------------------------------------------50
Block Andy
{
 Time 01:00
 +/- 00:10

 Pacing = Fast
```

```
 Class
 {
 }                                                                    60
 Keyword
 {
  Andy_Warhol
 }

 Not_Class
 {
 }
 Not_Keyword
 {                                                                    70
  Joey_Arias
 }
}
```

C.2 Complex Story Model

```
%__Complex story model of Joey becoming Andy

Block Joey-intro
{
 Time 00:10
 +/- 00:10

 Pacing = Slow

 Class                                                                10
 {
 }
 Keyword
 {
  greeting
```

```
  }

  Not_Class
  {
  }
  Not_Keyword
  {
  }
}

%-------------------------------------------------------------

Block Joey
{
  Time 01:05
  +/- 00:30

  Pacing = Slow

  Class
  {
  }
  Keyword
  {
    Joey_Arias
  }

  Not_Class
  {
  }
  Not_Keyword
  {
    Andy_Warhol
  }
}
```

```
%-------------------------------------------------------------

SubBlock Joey-no-wig
{
  Time 00:30
  +/- 00:10

  SuperBlock Joey
  Start 00:00                                                    60
  +/- 00:00

  Pacing = Slow

  Class
  {
  }
  Keyword
  {
  }                                                              70

  Not_Class
  {
  }
  Not_Keyword
  {
    wig
  }
}
%-------------------------------------------------------------80

SubBlock Joey-no-wig-pinning-hair
{
  Time 00:15
  +/- 00:10

  SuperBlock Joey-no-wig
```

```
Start 00:00
+/- 00:00

Pacing = Slow

Class
{
}
Keyword
{
  pinning_hair
}

Not_Class
{
}
Not_Keyword
{
}
}

%-----------------------------------------------------------

SubBlock Joey-no-wig-makeup
{
 Time 00:15
 +/- 00:10

 SuperBlock Joey-no-wig
 Start 00:15
 +/- 00:00

 Pacing = Slow

 Class
 {
```

```
 }
 Keyword
 {
  applying_make-up
 }

 Not_Class                                                    130
 {
 }
 Not_Keyword
 {
 }
}

%-------------------------------------------------------------

SubBlock Joey-putting-on-wig                                  140
{
 Time 00:10
 +/- 00:05

 SuperBlock Joey
 Start 00:30
 +/- 00:00

 Pacing = Slow
                                                              150
 Class
 {
 }
 Keyword
 {
  trying_on_wigs
 }

 Not_Class
```

```
  {                                                                    160
  }
  Not_Keyword
  {
  }
}

%-------------------------------------------------------------------

SubBlock Joey-with-wig
{                                                                      170
  Time 00:25
  +/- 00:05

  SuperBlock Joey
  Start 00:40
  +/- 00:00

  Pacing = Slow

  Class                                                                180
  {
  }
  Keyword
  {
    wig
  }

  Not_Class
  {
  }                                                                    190
  Not_Keyword
  {
  }
}
```

```
%-------------------------------------------------------------

SubBlock Joey-with-wig-spraying-wig
{
 Time 00:25                                                    200
 +/- 00:10

 SuperBlock Joey-with-wig
 Start 00:00
 +/- 00:00

 Pacing = Slow

 Class
 {                                                             210
 }
 Keyword
 {
  spraying_wig
 }

 Not_Class
 {
 }
 Not_Keyword                                                   220
 {
 }
}

%-------------------------------------------------------------

Block Joey-with-wig-spraying-wig-Sarah1
{
 Time 00:03
 +/- 00:01                                                     230
```

```
SuperBlock Joey-with-wig-spraying-wig
Start 00:10
+/- 00:00

Pacing = Slow

Class
{
}
Keyword
{
  Sarah
  watching
}

Not_Class
{
}
Not_Keyword
{
  Joey_Arias
}
}

%-------------------------------------------------------------

Block Joey-with-wig-spraying-wig-Sarah2
{
 Time 00:02
 +/- 00:01

 SuperBlock Joey-with-wig-spraying-wig
 Start 00:23
 +/- 00:00

 Pacing = Slow
```

```
 Class
 {                                                                    270
 }
 Keyword
 {
  Sarah
  watching
 }

 Not_Class
 {
 }                                                                    280
 Not_Keyword
 {
  Joey_Arias
 }
}

%-------------------------------------------------------------------

Block Joey->Andy
{                                                                     290
 Time 01:00
 +/- 00:20

 Pacing = Medium

 Class
 {
 }
 Keyword
 {                                                                    300
  Joey_Arias
  Andy_Warhol
 }
```

```
 Not_Class
 {
 }
 Not_Keyword
 {
 }                                                                    310
}

%-------------------------------------------------------------------

SubBlock Joey->Andy-no-sunglasses
{
 Time 00:20
 +/- 00:10

 SuperBlock Joey->Andy                                                320
 Start 00:00
 +/- 00:00

 Pacing = Medium

 Class
 {
 }
 Keyword
 {                                                                    330
 }

 Not_Class
 {
 }
 Not_Keyword
 {
  sunglasses
 }
```

```
}                                                                    340

%-------------------------------------------------------------------

SubBlock Joey->Andy-no-sunglasses-applying_eye_pencil
{
 Time 00:15
 +/- 00:10

 SuperBlock Joey->Andy-no-sunglasses
 Start 00:00                                                         350
 +/- 00:00

 Pacing = Slow

 Class
 {
 }
 Keyword
 {
  applying_eye_pencil                                                360
 }

 Not_Class
 {
 }
 Not_Keyword
 {
 }
}
                                                                     370

%-------------------------------------------------------------------

SubBlock Joey->Andy-putting-on-sunglasses
{
 Time 00:10
```

```
  +/- 00:10

  SuperBlock Joey->Andy
  Start 00:20
  +/- 00:00                                                          380

  Pacing = Medium

  Class
  {
  }
  Keyword
  {
    trying_on_sunglasses
  }                                                                  390

  Not_Class
  {
  }
  Not_Keyword
  {
  }
}

%-------------------------------------------------------------------400

SubBlock Joey->Andy-with-sunglasses
{
  Time 00:30
  +/- 00:20

  SuperBlock Joey->Andy
  Start 00:30
  +/- 00:00

                                                                     410

  Pacing = Medium
```

```
 Class
 {
 }
 Keyword
 {
  sunglasses
 }
                                                                    420
 Not_Class
 {
 }
 Not_Keyword
 {
 }
}

%-------------------------------------------------------------------

                                                                    430
SubBlock Joey->Andy-with-sunglasses-not-standing
{
 Time 00:10
 +/-  00:08

 SuperBlock Joey->Andy-with-sunglasses
 Start 00:00
 +/- 00:00

 Pacing = Medium                                                    440

 Class
 {
 }
 Keyword
 {
 }
```

```
 Not_Class
 {                                                                    450
 }
 Not_Keyword
 {
  standing
 }
}

%-------------------------------------------------------------------

SubBlock Joey->Andy-with-sunglasses-standing                          460
{
 Time 00:20
 +/- 00:10

 SuperBlock Joey->Andy-with-sunglasses
 Start 00:10
 +/- 00:00

 Pacing = Medium
                                                                      470
 Class
 {
 }
 Keyword
 {
  standing
 }

 Not_Class
 {                                                                    480
 }
 Not_Keyword
 {
```

```
  }
}

%-------------------------------------------------------------

Block Andy
{                                                              490
 Time 01:00
 +/- 00:10

 Pacing = Fast

 Class
 {
 }
 Keyword
 {                                                             500
  Andy_Warhol
 }

 Not_Class
 {
 }
 Not_Keyword
 {
  Joey_Arias
 }                                                             510
}
```

Appendix D

Homer Object Set Specifications

D.1 String Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <assert.h>

class String
{
 protected:
   int _size;
   char *_chararray;

  int addh (char x);
   // EFFECTS adds member x to the front end of TString.
   // REQUIRES x must be of the same type as TString.
   // MODIFIES TString.


  int addl (char x);
   // EFFECTS adds member x to the front end of TString.
   // REQUIRES x must be of the same type as TString.
   // MODIFIES TString.
```

```
char remh ();
  // EFFECTS  deletes one member from the tail end of TString.
  // MODIFIES TString.


char reml ();
  // EFFECTS  deletes one member from the front end of TString.
  // MODIFIES TString.

friend void delete_String (String *s)
 {
     delete (*s)._chararray;
     (*s)._chararray = NULL;
 }

public:


String ();
  // EFFECTS creates a new String object with no entries.
  // MODIFIES the String object.


~String ();
  // EFFECTS deletes String.
  // MODIFIES the String object.


String *Clear ();
  // EFFECTS clears the contents of a String, leaving the NULL String
  // MODIFIES the String object.


int empty();
  // EFFECTS returns 1 if String is empty, returns 0 otherwise.
```

```
int size();
  // EFFECTS returns the size of the TString.

char& operator[] (int i);
  // EFFECTS returns the character held in position String[i] of the
  //         the array of chars beginning at 1.
  // REQUIRES i <= String.size()

String &operator=(char *s);
  // EFFECTS  Allows a String object to be used as an lvalue for the
  //          = expression with char *s as an rvalue.  It converts this
  //          char * to a String and stores it in the String object.
  // MODIFIES the String object.

String &operator=(String& s);
  // EFFECTS  Allows a String object to be used as an lvalue for the
  //          = expression with String object as an rvalue.  It
  //          stores s in the String object.
  // MODIFIES the String object.

String *CharPtr2String (char *s);
  // EFFECTS  Converts *s to a String and stores it in the String object.
  // MODIFIES the String object.

char *String2CharPtr (char *c);
  // EFFECTS  Converts the String object and returns it as a char *.
  //          Note String2CharArray returns a pointer to a char *
  //          copy.
  // REQUIRES c must have suffiecient space allocated prior to S2CPtr.
```

```
  char *String2CharPtr ();
    // EFFECTS  Converts the String object and returns it as a char *.
    //          Note String2CharArray returns a pointer to the actual char *
    //          stored in the String object.  If a copy is desired use
    //          String2CharPtr (char *c) or use strcpy on this return val.
                                                                          100

  String *unparse (FILE *out);
    // EFFECTS prints String to out in a human readable form


};
```

D.2 Array Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "String/String.h"

class Array
{
  protected:
    int _size;
    String **_contents;

    friend void delete_Array (Array *a);

  public:

  Array ();
    // EFFECTS  creates a new Array object with no entries.
    // MODIFIES the Array object

  ~Array ();
    // EFFECTS deletes the Array object.
    // MODIFIES the Array object


  int empty();
    // EFFECTS returns 1 if the Array object contains no strings,
    //         returns 0 otherwise.


  int size();
    // EFFECTS returns the size of the Array object in number of strings.


  int addh (char *x);
```

```
// EFFECTS adds member x to the high end of the Array object.
//         returns the size of the Array object.
// MODIFIES the Array object.


int addl (char *x);                                                    40
// EFFECTS adds member x to the low end of the Array object.
//         returns the size of the Array object.
// MODIFIES the Array object.


int addh (String *y);
// EFFECTS adds member y to the high end of the Array object.
//         returns the size of the Array object.
// MODIFIES the Array object.
                                                                       50

int addl (String *y);
// EFFECTS adds member y to the low end of the Array object.
//         returns the size of the Array object.
// MODIFIES the Array object.


int remh (String *x);
// EFFECTS  deletes one member from the high end of the Array object.
//          inserts the object in x (NULL String if empty)            60
//          and returns the size of the Array after deletion.
// MODIFIES the Array object.


int reml (String *x);
// EFFECTS  deletes one member from the low end of the Array object.
//          inserts the object in x (NULL String if empty)
//          and returns the size of the Array after deletion.
// MODIFIES the Array object.
                                                                       70
```

```
int remh ();
  // EFFECTS  deletes one member from the high end of the Array object.
  //          and returns the size of the Array after deletion.
  // MODIFIES the Array object.


int reml ();
  // EFFECTS  deletes one member from the low end of the Array object.
  //          and returns the size of the Array after deletion.
  // MODIFIES the Array object.


String &operator[] (int i);
  // EFFECTS  allows the Array to be referenced by index as a standard
  //          c array.  Ex.  s[1], s[2].  Note Array objects have their
  //          first member in the number 1 position.  Ex. s[1].
  //          returns the String object referenced by i and may be
  //          used as an lvalue.


 unparse (FILE *out);
   // EFFECTS  prints the Array object to out in a human readable form
};
```

D.3 AlphaArray Object

```
#include <strings.h>
#include <stdio.h>
#include "Array/Array.h"

extern "C" int strcasecmp(const char *, const char *);

class AlphaArray : private Array
{
private:

  friend void delete_AlphaArray (AlphaArray *a);

public:

  AlphaArray () : Array()
    // EFFECTS creates a new Alphaarray object with no entries
    // MODIFIES the Alphaarray object
    {}

  ~AlphaArray ()
    // EFFECTS deletes the Alphaarray object
    // MODIFIES the Alphaarray object
    {}

  int empty();
    // EFFECTS returns 1 if the Alphaarray object has no entries,
    //         returns 0 otherwise.

  int size();
    // EFFECTS returns the number of entries in the Alphaarray object.
```

```
int ADd (char *s);
  // EFFECTS adds member s to AlphaArray in alphabetical order. Note
  //         duplicates are not added.
  // MODIFIES the Alphaarray object.

int ADd (String *str);
  // EFFECTS adds member s to AlphaArray in alphabetical order. Note
  //         duplicates are not added.
  // MODIFIES the Alphaarray object.

int DEl (char *s);
  // EFFECTS deletes member s from AlphaArray.
  // MODIFIES the Alphaarray object.

int DEl (String *str);
  // EFFECTS deletes member s from AlphaArray.
  // MODIFIES the Alphaarray object.

String& operator[] (int i);
  // EFFECTS  allows the Alphaarray to be refernced by index as a standard
  //          c array.  Ex.  aa[1], aa[2].  Note Alphaarray objects have
  //          their first member in the number 1 position.  Ex. aa[1].
  //          if [] is used as an lvalue alphabetical order cannot be
  //          maintained..

unparse (FILE *out);
  // EFFECTS  prints the Alphaarray object to out in a human readable
  //          form.
```

```
  unparse ();
    // EFFECTS  prints the Alphaarray object to out in a human readable
    //          form.

};
```

D.4 StrataLine Object

```
#include <strings.h>
#include <stdio.h>
#include "AlphaArray/AlphaArray.h"
#define MAX_KEY 50

class StrataLine
{
 protected:
  String volume;
  int _in, _out;
  int _content, _speed;
  String title;
  int sort_val;

  friend delete_StrataLine (StrataLine *s)
   {
     void delete_String (String *);
     void delete_AlphaArray (AlphaArray *);

     delete_String (&(*s).volume);
     delete_String (&(*s).title);
     delete_AlphaArray (&(*s).Class_Keyword);
   }

public:

 AlphaArray Class_Keyword;

 StrataLine();
   // EFFECTS  creates a new StrataLine object
   // MODIFES  the StrataLine object


 ~StrataLine();
```

```
  // EFFECTS  deletes the StrataLine object
  // MODIFES  the StrataLine object


StrataLine *Clear ();
  // EFFECTS clears the contents of a StrataLine, leaving the StrataLine
  //         int the newly created state.
  // MODIFIES the String object.


int AddCKPair (char *c, char *key);
  // EFFECTS  adds the class/keyword pair to the list of pairs
  //          for the StrataLine object. Note, no dups
  //          returns the number of c/k pairs after the Add op.
  // MODIFIES the StrataLine object (Class_Keyword A-Array)


int DelCKPair (char *c, char *key);
  // EFFECTS  deletes the class/keyword pair from the list
  //          of pairs for the StrataLine object.
  //          returns the number of c/k pairs after the Del op.
  // MODIFIES the StrataLine object (Class_Keyword A-Array)


SetSortVal (int s);
  // EFFECTS  Puts the value of s into the sort value field of
  //          the StrataLine object.  A StoryLineArray can be
  //          sorted in ascending order by this value
  // MODIFES  the StrataLine object


int GetSortVal ();
  // EFFECTS  returns the value of the sort value field
  //          of the StrataLine object
```

```
SetVolume (char *vol);
  // EFFECTS  Puts a copy of the string at vol into the volume field of
  //          the StrataLine object.
  // MODIFES  the StrataLine object


char *GetVolume ();
  // EFFECTS  returns a copy of the string in the volume field
  //          of the StrataLine object                                  80


SetTitle (char *tit);
  // EFFECTS  Puts a copy of the string at tit into the title field of
  //          the StrataLine object.
  // MODIFES  the StrataLine object


char *GetTitle ();
  // EFFECTS  returns a copy of the string in the title field
  //          of the StrataLine object                                  90


SetIn (int in);
  // EFFECTS  Puts the value of in into the in point field of
  //          the StrataLine object.
  // MODIFES  the StrataLine object


int GetIn ();
  // EFFECTS  returns the value of the in point field                  100
  //          of the StrataLine objec


SetOut (int out);
  // EFFECTS  Puts the value of out into the out point field of
  //          the StrataLine object.
```

```
  // MODIFES  the StrataLine object


int GetOut ();                                                          110
  // EFFECTS  returns the value of the out point field
  //          of the StrataLine objec


SetContent (int cont);
  // EFFECTS  Puts the value of cont into the content field of
  //          the StrataLine object.
  // MODIFES  the StrataLine object

                                                                        120
int GetContent ();
  // EFFECTS  returns the value of the content field
  //          of the StrataLine objec


SetSpeed (int speed);
  // EFFECTS  Puts the value of speed into the speed field of
  //          the StrataLine object.
  // MODIFES  the StrataLine object
                                                                        130

int GetSpeed ();
  // EFFECTS  returns the value of the in speed field
  //          of the StrataLine objec


StrataLine *Copy (StrataLine *s1);
  // EFFECTS  places a copy of StrataLine s1 in the current
  //          StrataLine object.
  // MODIFIES the StrataLine object.                                    140

unparse (FILE *out);
```

```
    // EFFECTS  prints the StrataLine object to out in a human readable form

};
```

D.5 StrataFile Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "StrataLine/StrataLine.h"

class StrataFile
{
  protected:
    int _size;
    StrataLine **_contents;                                          10

    friend void delete_StrataFile (StrataFile *s)
        {
          int i;
          for (i=1; i <= (*s)._size; i++)
            delete_StrataLine (((*s)._contents)[i]);
          for (i=1; i <= (*s)._size; i++)
            delete (((*s)._contents)[i]);
          delete (*s)._contents;
        }                                                            20

  public:

  StrataFile ();
    // EFFECTS  creates a new StrataFile object with no entries.
    // MODIFIES the StrataFile object

  ~StrataFile ();
    // EFFECTS deletes the StrataFile object.
    // MODIFIES the StrataFile object                                30


  int empty();
    // EFFECTS returns 1 if the StrataFile object contains no StrataLines,
```

```
//          returns 0 otherwise.


int size();
 // EFFECTS returns the size of the StrataFile object in number of StrataLines.
                                                                              40


int addh (StrataLine *y);
 // EFFECTS adds member x to the high end of the StrataFile object.
 //         returns the size of the StrataFile object.
 // MODIFIES the StrataFile object.


int addl (StrataLine *y);
 // EFFECTS adds member x to the low end of the StrataFile object.
 //         returns the size of the StrataFile object.                       50
 // MODIFIES the StrataFile object.


int remh (StrataLine *x);
 // EFFECTS  deletes one member from the high end of the StrataFile object.
 //          inserts the deleted object in x and
 //          returns the size of the StrataFile after deleteion.
 // MODIFIES the StrataFile object.

                                                                              60
int reml (StrataLine *x);
 // EFFECTS  deletes one member from the low end of the StrataFile object.
 //          inserts the deleted object in x and
 //          returns the size of the StrataFile after deleteion.
 // MODIFIES the StrataFile object.


int remh ();
 // EFFECTS  deletes one member from the high end of the StrataFile object.
 //          returns the size of the StrataFile after deleteion.             70
```

```
  // MODIFIES the StrataFile object.


int reml ();
  // EFFECTS  deletes one member from the low end of the StrataFile object.
  //          returns the size of the StrataFile after deletcion.
  // MODIFIES the StrataFile object.


int ADd (StrataLine *str);                                              80
  // EFFECTS adds member s to StrataFile in ascending order of sort_val.
  // MODIFIES the StrataFile object.


StrataLine &operator[] (int i);
  // EFFECTS  allows the StrataFile to be referaced by index as a standard
  //          c StrataFile.  Ex.  s[1], s[2].  Note StrataFile objects have their
  //          first member in the number 1 position.  Ex. s[1].
  //          returns the StrataLine object refernced by i and may be
  //          used as an lvalue.                                        90


unparse (FILE *out);
  // EFFECTS  prints the StrataFile object to out in a human readable form
};


                                                                        100
```

D.6 StoryLine Object

```
#include <stdio.h>
#include "StrataFile/StrataFile.h"


class StoryLine
{
 protected:
  String _b1;
  String _b2;
  String _temperament;                                          10
  Array l;

  /* types:
       KEY
       CLASS
  */

  friend void delete_StoryLine (StoryLine *s)
    {
      void delete_String (String *);                            20
      void delete_Array (Array *);

      delete_String (&(*s)._b1);
      delete_String (&(*s)._b2);
      delete_String (&(*s)._temperament);
      delete_Array (&(*s).l);
    }

 public:
                                                                30

  StoryLine();
    // EFFECTS  creates a new Storyline object
    // MODIFIES the StoryLine object
```

```
~StoryLine()
 // EFFECTS  deletes the Storyline object
 // MODIFIES the StoryLine object
 {
 };

void Clear ();
 // EFFECTS  Sets all the values in the StoryLine object
 //          to their creating defaults.
 // MODIFIES the StoryLine object

void SetNegative ();
 // EFFECTS  Sets the Storyline object's temperament to negative
 // MODIFIES the StoryLine object

void SetPositive ();
 // EFFECTS  Sets the Storyline object's temperament to positive
 // MODIFIES the StoryLine object

char *GetTemperament();
 // EFFECTS  Returns the temperament (POS or NEG) of the StoryLine
 //          object as a String.

int SetFromBlock(char *b1);
 // EFFECTS  Sets block1 of the Storyline object to b1.
 // MODIFIES the StoryLine object

int SetFromBlock(String *b1);
 // EFFECTS  Sets block1 of the Storyline object to b1.
```

```
  // MODIFIES the StoryLine object


char *GetFromBlock();
  // EFFECTS  Returns block1 of the Storyline as a String.


int SetToBlock(char *b2);
  // EFFECTS  Sets block2 of the Storyline object to b2.
  // MODIFIES the StoryLine object                                    80


int SetToBlock(String *b2);
  // EFFECTS  Sets block2 of the Storyline object to b2.
  // MODIFIES the StoryLine object


char *GetToBlock();
  // EFFECTS  Returns block2 of the Storyline as a String.
                                                                      90


void SetFromValues (char *type, char *w);
  // EFFECTS  Set the start attributes of the StoryLine.  The start
  //          of the StoryLine is always at block1.  type specifies
  //          either CLASS or KEYWORD.  w specifies the actual member
  //          of type.  Both type and w are stored in the StoryLine object.
  // MODIFIES the StoryLine object

char *GetFromType ();
  // EFFECTS  Returns the Storyline data type for the from block      100
  //          as a String.

char *GetFromWord ();
  // EFFECTS  Returns the Storyline word value for the from block
  //          as a String.
```

```
void SetToValues (char *type, char *w);
  // EFFECTS  Set the end attributes of the StoryLine.  The end
  //          of the StoryLine is always at block2.  type specifies
  //          either CLASS or KEYWORD.  w specifies the actual member
  //          of type.  Both type and w are stored in the StoryLine object.
  // MODIFIES the StoryLine object

char *GetToType ();
  // EFFECTS  Returns the Storyline data type for the to block
  //          as a String.

char *GetToWord ();
  // EFFECTS  Returns the Storyline word value for the to block
  //          as a String.


StoryLine *Copy (StoryLine *S1);
  // EFFECTS  Copies the values in S1 to the StoryLine objecy
  // MODIFIES The StoryLine object.


void unparse (FILE *out);
  // EFFECTS  prints the StoryLine object to out in form readable
  //          by Homer. (semi-human readable/computer readable)


void Unparse (FILE *out);
  // EFFECTS  prints the StoryLine object to out in a human readable
  //          form

};
```

D.7 Plot Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "StoryLine/StoryLine.h"

class Plot
{
 protected:
   int _size;
   StoryLine **_contents;                                          10

 public:

 Plot ();
  // EFFECTS  creates a new Plot object with no entries.
  // MODIFIES the Plot object

 ~Plot ();
  // EFFECTS deletes the Plot object.
  // MODIFIES the Plot object                                      20


 int empty();
  // EFFECTS returns 1 if the Plot object contains no StoryLines,
  //         returns 0 otherwise.


 int size();
  // EFFECTS returns the size of the Plot object in number of StoryLines.
                                                                  30

 int addh (StoryLine *y);
  // EFFECTS adds member x to the high end of the Plot object.
  //         returns the size of the Plot object.
```

```
// MODIFIES the Plot object.


int addl (StoryLine *y);
// EFFECTS adds member x to the low end of the Plot object.
//         returns the size of the Plot object.                    40
// MODIFIES the Plot object.


int remh (StoryLine *x);
// EFFECTS  deletes one member from the high end of the Plot object.
//          inserts the deleted StoryLine object in x
//          and returns the size of the Plot after deletion.
// MODIFIES the Plot object.

                                                                    50
int reml (StoryLine *x);
// EFFECTS  deletes one member from the low end of the Plot object.
//          inserts the deleted StoryLine object in x
//          and returns the size of the Plot after deletion.
// MODIFIES the Plot object.


int remh ();
// EFFECTS  deletes one member from the high end of the Plot object.
//          and returns the size of the Plot after deletion.        60
// MODIFIES the Plot object.


int reml ();
// EFFECTS  deletes one member from the low end of the Plot object.
//          and returns the size of the Plot after deletion.
// MODIFIES the Plot object.


StoryLine &operator[] (int i);                                      70
```

```
  // EFFECTS  allows the Plot to be refernced by index as a standard
  //          c Plot.  Ex.  s[1], s[2].  Note Plot objects have their
  //          first member in the number 1 position.  Ex. s[1].
  //          returns the StoryLine object refernced by i and may be
  //          used as an lvalue.


 unparse (FILE *out);
  // EFFECTS  prints the Plot object to out in a human readable form
};                                                                    80
```

D.8 Block Object

```
#include "Plot/Plot.h"
#define MAX_SUB_BLOCKS 25

class Block
{
 protected:

  String name;

  int min;
  int sec;

  int delta_min;
  int delta_sec;

  int sb_min;
  int sb_sec;

  int sb_delta_min;
  int sb_delta_sec;

  String pacing;

  String spb;

  friend void delete_Block (Block *b);

 public:

  Array sbb;
  // array of sub-block names

  AlphaArray classes;
  AlphaArray keys;
```

```
AlphaArray not_classes;
AlphaArray not_keys;

Plot Block_Plot;
StrataFile Block_Strata;                                        40

Block ();
  // EFFECTS  creates a new Block object
  // MODIFES  the Block object


~Block ();
  // EFFECTS  deletes the Block object
  // MODIFES  the Block object
                                                                50
Clear ();
  // EFFECTS  sets all values in the Block object equal to the
  //          default creation values.
  // MODIFES  the Block object

SetName (char *s);
  // EFFECTS  Puts a copy of the string at s into the name field of
  //          the Block object
  // MODIFES  the Block object
                                                                60
char *GetName ();
  // EFFECTS  returns the string most recently stored in the name field
  //          of the Block object


SetTime (int m, int s);
  // EFFECTS  Stores the time of the Block object in m minutes and
  //          s seconds.
  // MODIFES  the Block object                                  70
```

```
int GetTime ();
  // EFFECTS  Returns the time of the Block object
  //          in seconds.


SetDelta (int m, int s);
  // EFFECTS  Stores the +/- time of the Block object in m minutes and
  //          s seconds.                                                  80
  // MODIFES  the Block object


int GetDelta ();
  // EFFECTS  Returns the +/- time of the Block object
  //          in seconds.


SetSBTime (int m, int s);
  // EFFECTS  Stores the super-Block start time of the Block object      90
  //          in m minutes and s seconds.
  // MODIFES  the Block object


int GetSBTime ();
  // EFFECTS  Returns the super-Block start time of the Block object
  //          in seconds.


SetSBDelta (int m, int s);                                             100
  // EFFECTS  Stores the super-Block start +/- time of the Block object
  //          in m minutes and s seconds.
  // MODIFES  the Block object


int GetSBDelta ();
```

```
// EFFECTS  Returns the super-Block start +/- time of the Block object
//          in seconds.

                                                                    110
SetPacing (char *s);
// EFFECTS  Puts a copy of the string at s into the pacing field of
//          the Block object
// MODIFES  the Block object

char *GetPacing ();
// EFFECTS  returns the string most recently stored in the pacing field
//          of the Block object
                                                                    120
int AddClass (char *s);
// EFFECTS  Adds a copy of the string at s into the Class
//          array of the Block object, in alphabetical order
//          without duplicates
// MODIFES  the Block object

int AddKey (char *s);
// EFFECTS  Adds a copy of the string at s into the Key               130
//          array of the Block object, in alphabetical order
//          without duplicates
// MODIFES  the Block object

int AddNotClass (char *s);
// EFFECTS  Adds a copy of the string at s into the NotClass
//          array of the Block object, in alphabetical order
//          without duplicates
// MODIFES  the Block object                                          140
```

```
int AddNotKey (char *s);
  // EFFECTS  Adds a copy of the string at s into the NotKey
  //          array of the Block object, in alphabetical order
  //          without duplicates
  // MODIFES  the Block object


int DelClass (char *s);                                          150
  // EFFECTS  Removes the string s from the Class
  //          array of the Block object
  // MODIFES  the Block object


int DelKey (char *s);
  // EFFECTS  Removes the string s from the Key
  //          array of the Block object
  // MODIFES  the Block object
                                                                 160

int DelNotClass (char *s);
  // EFFECTS  Removes the string s from the NotClass
  //          array of the Block object
  // MODIFES  the Block object


int DelNotKey (char *s);
  // EFFECTS  Removes the string s from the NotKey
  //          array of the Block object                         170
  // MODIFES  the Block object


SetSuperBlockName (char *s);
  // EFFECTS  Puts a copy of the string at s into the super-Block name
  //          field of the Block object
  // MODIFES  the Block object
```

```
char *GetSuperBlockName ();                                          180
  // EFFECTS  Returns a copy of the string at the super-Block name
  //          field of the Block object


Block *Copy (Block *b1);
  // EFFECTS places a copy of Block b1 in the current Block object
  // MODIFIES the Block object


unparse (FILE *out);                                                 190
  // EFFECTS  prints the Block object to out in a human readable form
};
```

D.9 BlockArray Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "Block/Block.h"

class BlockArray
{
  protected:
    int _size;
    Block **_contents;                                          10

  public:

  BlockArray ();
    // EFFECTS  creates a new BlockArray object with no entries.
    // MODIFIES the BlockArray object

  ~BlockArray ();
    // EFFECTS deletes the BlockArray object.
    // MODIFIES the BlockArray object                          20


  int empty();
    // EFFECTS returns 1 if the BlockArray object contains no Blocks,
    //         returns 0 otherwise.


  int size();
    // EFFECTS returns the size of the BlockArray object in number of Blocks.
                                                                30

  int addh (Block *y);
    // EFFECTS adds member x to the high end of the BlockArray object.
    //         returns the size of the BlockArray object.
```

```
  // MODIFIES the BlockArray object.


int addl (Block *y);
  // EFFECTS adds member x to the low end of the BlockArray object.
  //         returns the size of the BlockArray object.                    40
  // MODIFIES the BlockArray object.


int remh (Block *x);
  // EFFECTS  deletes one member from the high end of the BlockArray object.
  //          inserts the deleted Block object in x
  //          and returns the size of the BlockArray after deletion.
  // MODIFIES the BlockArray object.

                                                                            50
int reml (Block *x);
  // EFFECTS  deletes one member from the low end of the BlockArray object.
  //          inserts the deleted Block object in x
  //          and returns the size of the BlockArray after deletion.
  // MODIFIES the BlockArray object.


int remh ();
  // EFFECTS  deletes one member from the high end of the BlockArray object.
  //          and returns the size of the BlockArray after deletion.        60
  // MODIFIES the BlockArray object.


int reml ();
  // EFFECTS  deletes one member from the low end of the BlockArray object.
  //          and returns the size of the BlockArray after deletion.
  // MODIFIES the BlockArray object.


Block &operator[] (int i);                                                  70
```

```
  // EFFECTS  allows the BlockArray to be refernced by index as a standard
  //          c BlockArray.  Ex.  s[1], s[2].  Note BlockArray objects have their
  //          first member in the number 1 position.  Ex. s[1].
  //          returns the Block object refernced by i and may be
  //          used as an lvalue.


 unparse (FILE *out);
  // EFFECTS  prints the BlockArray object to out in a human readable form
};                                                                          80
```

D.10 Report Object

```
#include "BlockArray/BlockArray.h"

class Report
{
 protected:

  String name;

  int min;
  int sec;

  int delta_min;
  int delta_sec;

  int tot_min;
  int tot_sec;

  int no_segs;

  friend void delete_Report (Report *b);

 public:

  Report ();
    // EFFECTS  creates a new Report object
    // MODIFES  the Report object


  ~Report ();
    // EFFECTS  deletes the Report object
    // MODIFES  the Report object

  Clear ();
    // EFFECTS  sets all values in the Report object equal to the
```

```
//          default creation values.
// MODIFES  the Report object

SetName (char *s);
// EFFECTS  Puts a copy of the string at s into the name field of
//          the Report object                                          40
// MODIFES  the Report object


char *GetName ();
// EFFECTS  returns the string most recently stored in the name field
//          of the Report object


SetTime (int m, int s);
// EFFECTS  Stores the time of the Report object in m minutes and     50
//          s seconds.
// MODIFES  the Report object


int GetTime ();
// EFFECTS  Returns the time of the Report object
//          in seconds.


SetDelta (int m, int s);                                              60
// EFFECTS  Stores the +/- time of the Report object in m minutes and
//          s seconds.
// MODIFES  the Report object


int GetDelta ();
// EFFECTS  Returns the +/- time of the Report object
//          in seconds.

                                                                      70
```

```
SetTotTime (int m, int s);
  // EFFECTS  Stores the total time of footage available to object name
  //          in the Report object in m minutes and
  //          s seconds.
  // MODIFES  the Report object


int GetTotTime ();
  // EFFECTS  Returns the tot time of footage available to the Report object
  //          in seconds.                                                    80


SetNoSegments (int n);
  // EFFECTS  Stores the number of segements of footage available to
  //          name in the Report object.
  // MODIFES  the Report object


int GetNoSegments ();
  // EFFECTS  Returns the number of segments of footage available to        90
  //          to name in the Report object.
  //          in seconds.


AddStrataLine (StrataLine *s);
  // EFFECTS  Modifies the Report object to reflect s being footage
  //          available to the Report object.  Updates total time and
  //          number of segments of footage available.
  // MODIFIES the Reoprt object.
                                                                          100


Report *Copy (Report *b1);
  // EFFECTS places a copy of Report b1 in the current Report object
  // MODIFIES the Report object
```

```
  unparse (FILE *out);
    // EFFECTS prints the Report object to out in a human readable form
};
```

110

D.11 ReportArray Object

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "Report/Report.h"

class ReportArray
{
  protected:
    int _size;
    Report **_contents;                                          10

  public:

  ReportArray ();
    // EFFECTS  creates a new ReportArray object with no entries.
    // MODIFIES the ReportArray object

  ~ReportArray ();
    // EFFECTS deletes the ReportArray object.
    // MODIFIES the ReportArray object                          20


  int empty();
    // EFFECTS returns 1 if the ReportArray object contains no Reports,
    //         returns 0 otherwise.


  int size();
    // EFFECTS returns the size of the ReportArray object in number of Reports.
                                                                 30

  int addh (Report *y);
    // EFFECTS adds member x to the high end of the ReportArray object.
    //         returns the size of the ReportArray object.
```

```
// MODIFIES the ReportArray object.


int addl (Report *y);
// EFFECTS adds member x to the low end of the ReportArray object.
//         returns the size of the ReportArray object.
// MODIFIES the ReportArray object.


int remh (Report *x);
// EFFECTS  deletes one member from the high end of the ReportArray object.
//          inserts the deleted Report object in x
//          and returns the size of the ReportArray after deletion.
// MODIFIES the ReportArray object.


int reml (Report *x);
// EFFECTS  deletes one member from the low end of the ReportArray object.
//          inserts the deleted Report object in x
//          and returns the size of the ReportArray after deletion.
// MODIFIES the ReportArray object.


int remh ();
// EFFECTS  deletes one member from the high end of the ReportArray object.
//          and returns the size of the ReportArray after deletion.
// MODIFIES the ReportArray object.


int reml ();
// EFFECTS  deletes one member from the low end of the ReportArray object.
//          and returns the size of the ReportArray after deletion.
// MODIFIES the ReportArray object.


Report &operator[] (int i);
```

```
  // EFFECTS  allows the ReportArray to be refernced by index as a standard
  //          c ReportArray.  Ex.  s[1], s[2].  Note ReportArray objects have their
  //          first member in the number 1 position.  Ex. s[1].
  //          returns the Report object refernced by i and may be
  //          used as an lvalue.


 unparse (FILE *out);
  // EFFECTS  prints the ReportArray object to out in a human readable form
};                                                                          80
```

D.12 Story Object

```
#include <stdio.h>
#include "ReportArray/ReportArray.h"


class Story
{
 protected:

 public:
                                                                    10
  StrataFile Strata;
  BlockArray Blocks;
  Plot StoryLines;
  ReportArray Reports;

  Story();
    // EFFECTS  creates a new Story object
    // MODIFIES the Story object


                                                                    20
  ~Story();
    // EFFECTS  deletes the Story object
    // MODIFIES the Story object



};
```

Appendix E

Homer Object Set Code

Source code available by special request only.

Appendix F

Homer Control Code

Source code available by special request only.

Bibliography

[1] Thomas G. Aguierre Smith. Stratification: Toward a computer representation of the moving image. Technical report, The Media Lab, MIT, 1991.

[2] D. Applebaum. The galatea network video device control system. Technical report, The Media Lab, MIT, 1989.

[3] Giles R. Bloch. From concepts to film sequences. Technical report, Yale University, Department of Computer Science, Artificial Intellingence Lab, 1900.

[4] Segei Eisenstein. *The Film Sense.* Harcourt Brace and Company, 1942.

[5] Segei Eisenstein. *Film Form.* Harcourt Brace and Company, 1949.

[6] Benjamin Rubin. Constrain-based cinematic editing. Master's thesis, Massachusetts Institute of Technology, June 1989.