High-level scripting environments for interactive multimedia systems

by
Stefan Panayiotis Agamanolis

B.A., Computer Science
Oberlin College, Oberlin, Ohio
May 1994

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology
February 1996

Signature of Author:

Program in Media Arts and Sciences
19 January 1996

Certified by:

V. Michael Bove, Jr.
Associate Professor of Media Technology
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by:

Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

High-level scripting environments for interactive multimedia systems

by
Stefan Panayiotis Agamanolis

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on January 19, 1996
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

Abstract

Interactive multimedia systems typically consist of many kinds of media objects and input and output components, all of which must be controlled in real time to form a presentation. Although nothing can replace human to human interaction, in many cases it is desirable to automate these management tasks. As the complexity of such systems grows, some method of *scripting* these presentations in a high-level manner becomes a necessity.

Current solutions are inadequate. Ultra-high-level visual (non-scripted) authoring systems restrict the creativity of the artist, whereas programming in a low-level general-purpose language forces the artist to become a computer scientist. Many systems that lie between these two extremes handle interactivity in a very primitive manner. This thesis details work on the design and implementation of an intuitive and elegant scripting environment that does not sacrifice expressivity or extensibility. In addition, a real-time interactive object-based audio and video system has been created to operate under this new environment.

Thesis Supervisor: V. Michael Bove, Jr.
Title: Associate Professor of Media Technology

This work was supported by the Television of Tomorrow consortium.

High-level scripting environments for interactive multimedia systems

by

Stefan Panayiotis Agamanolis

The following people served as readers for this thesis:

Reader:

Glorianna Davenport
Associate Professor of Media Technology
Program in Media Arts and Sciences

Reader:

Dave L. Sprague
Manager, Multimedia Architecture
Intel Corporation
Media Laboratory Research Affiliate

Acknowledgments

Many people deserve special recognition for the support they have given me over the duration of this work...

First and foremost, my research advisor, Mike Bove, for giving me the opportunity to work at the Media Lab. His ideas and down-to-earth suggestions have guided and motivated me throughout my time here.

Glorianna Davenport, for her invaluable thoughts on the artistic side of my work. In many ways, she was my second research advisor.

Dave Sprague, for his time in reading my thesis and for giving me an outside point of view.

Master system administrator, Henry Holtzman, for keeping the garden computers and software working perfectly and for helping me set up the computer in my office.

Araz Inguilizian, for creating the innovative object-based audio software that I use in my system.

Shawn Becker, for helping me dissect the internals of the Cheops library routines, for creating cool three-dimensional models, and for being a great friend.

John Watlington, for keeping Cheops working and for helping me understand its inner workings.

Teresa Chang, Brett Granger, and Katy Brown, for saving me hours of work by revealing their discoveries about the oddities of Cheops.

Tony Romain, for being a real director, and Rachel Lillis and Michael Slayton, for being real actors.

Everyone else involved in the highly successful *Wallpaper* shoot, including Eugene Lin, David Tamés, Andrew Beechum, and Bill Butera.

Richard Salter, Rhys Price Jones, Bob Geitz, and Garold Garrett, for challenging me and for instilling in me an enthusiasm for learning that will always be a part of me.

Dan Gruhl, for setting up and maintaining the LaTeX typesetter in the garden.

My officemate and good friend, Chris Verplaetse, for making me laugh and for keeping me sane.

David Letterman, for entertaining me every night of the past year and a half.

My housemates, Martin Szummer, Matt Krom, and April Paffrath, for being awesome people and for supporting me through the last part of my work.

And finally, my parents, my sister Chrissie, and my cats, Whiskers, Frisky, and Saucy, for allowing me to become the greatest power in the universe.

Contents

List of Figures

Chapter 1

Introduction

The word "multimedia" has become so dreadfully hackneyed from overuse that it is difficult to determine what it truly means or if it has even been invented yet. In the simplest possible terms, it refers to a combination of many different kinds of individual media for a common purpose, but this meaning is often twisted. Throwing together a big-screen video monitor, a couple of slide projectors, and a deafening sound system might produce a compelling multimedia presentation, but that only acknowledges the physical devices that are used to render the various media objects. A better way to think about multimedia is as a melding of several different *conceptual* media forms ranging from textual to auditory to graphical objects. These categories could also be subdivided further—graphical media could be divided into static objects and animated objects, and these objects could be two- or three-dimensional in nature. Everything from a simple black-and-white photograph to a CAD model of a skyscraper could be considered graphical media. The advantage of this interpretation of multimedia is that it is completely independent from the way these media forms are displayed or otherwise presented. A three-dimensional media object might be rendered in a small picture in a book or on an interactive holographic video display.

Under this definition, everything from a tiny coupon in the Sunday paper to a multi-million dollar rock concert would be considered a multimedia presentation. These two examples also have one important thing in common: they are both *static* entities, meaning that these

productions are not tailored in any way to the viewer's internal state or personal traits, nor do they allow her to directly interact in any way to change the presentation. She simply experiences the same exact thing from one "viewing" to the next. (Of course she could tear up the coupon or throw tomatoes at the concert performers and the presentations would probably change, but neither of them has any *intended* points of interaction.)

This is where the line can be drawn between *static* and *dynamic* multimedia. Dynamic multimedia might be roughly classified as either *interactive* or *personalized.* The line is very fuzzy, but in general, *interactive* media presentations are those in which the viewer or listener (or simply the *experiencer*) takes an *active* part in modifying the media delivery in some way, whereas in *personalized* media, the experiencer only *passively* affects the presentation, perhaps if the delivery system knows something about the viewing conditions or if it learns something about the experiencer's personality or habits. Examples of interactive multimedia presentations are pop-up books, board games, "interactive movies", most kinds of live shows, and flight simulators. On the other hand, electronic newspapers, junk-mail sweepstakes and advertisements, and world wide web documents often incorporate various degrees of personalization in their presentations.

Concentrating on interactivity for the moment, some helpful distinctions can be made between different kinds of interaction, such as between cases where it is highly *restricted* and cases where it is virtually *unlimited.* Interaction can be termed *restricted* if at any point the experiencer can only pick from a small set of choices which direction the presentation will take. For example, if a magician is performing a card trick and asks a member of the audience to choose a card from the deck, the audience member only has 52 discrete choices in that interaction—she can't choose to pick two or three cards, nor could she choose to take *all* the cards and throw them up in the air. Most interactive multimedia presentations in existence today can probably be categorized as restricted. However, it is also possible to imagine a few cases where interaction is practically *unlimited.* For example if a television viewer calls up a talk show and is put on the air, that viewer has the ability to say just about anything and the talk show host or producers will most likely be prepared to react in an appropriate way no matter what is said.

A more useful distinction can be drawn between situations where the interaction is *continuous* and where it is only *intermittent.* In some of the latest so-called "interactive" movies, interaction only consists of occasionally pressing buttons to pick from a small set of choices what will happen next in the story. Between these key points, of which there might be very few, the presentation is purely static, playing out just like a regular movie. On the other hand, the involvement of a trainee using a sophisticated flight simulator is not at all intermittent—the presentation provides for and depends on continuous and almost unlimited interaction.

Perhaps the ultimate interactive multimedia presentations are live one-on-one situations where the presenter is delivering structured media that appeal to all five senses for an organized purpose, and where modifications are made based on the passive or active responses of the experiencer. Although nothing can replace this kind of human to human interaction, dynamically changing a multimedia presentation based on real-time input is a process that is usually desirable to automate. Of course, there are many cases where automation would be either impossible or unthinkable, but computers are very good at controlling physical devices which might render the media objects that need to be presented, and they can respond to interaction almost instantly.

In non-interactive situations, it is possible to exactly "script" every aspect of the presentation from start to end. If this script is written in a machine-readable format, a computer could control the entire production by simply sending the right signals to the right components (machine or human) at the appropriate times. In cases where interaction is only intermittent, it might be possible to script everything between those points of interaction. However, the problem becomes extremely complex in cases where continuous dynamic interaction is desired. In the flight simulator example, a specialized computer program would have to be written to monitor the input from the user and control all the output devices in the simulator.

To make this scripting job simpler, very often some kind of special "scripting language" is used to allow an author to express high level ideas and processes in a format the computer

can interpret to run a production. Languages for controlling static or minimally interactive media presentations can be relatively simple since there is not much processing necessary beyond sending preset signals at preset times. However, fully interactive systems usually demand much more processing and language expressivity, more than can be handled by a simple scripting language. Very often, applications of this type (like complicated simulators or three-dimensional virtual reality systems) are necessarily written at a much lower level in programming languages like C, and they are usually tailored for specific productions and physical processing, input, and output hardware. In these cases, scripting ceases to be scripting and becomes laborious computer software hacking.

Is there some middle ground that can be reached where it would be possible to script complex interactive presentations in a platform-independent high-level manner without sacrificing efficiency and expressivity?

Chapter 2

Previous efforts

Past work in this area has been quite moderate. There have been many systems and languages designed for controlling static presentations. Their function is basically to organize events in chronological order and to execute corresponding instructions to create the desired output. However, many of these languages are not very extensible—the user's ability to extend the functionality of the language is highly restricted or extremely difficult. The languages that have flourished are those that allow users to add their own creative elements or to simplify the script-writing process by writing procedures or creating similar abstractions. More recent systems are often tailored for certain kinds of applications, such as interactive television, informational kiosks, or computer-based training, and they usually give the ability to express some kind of interactivity, but unfortunately it is often very minimal.

2.1 Models of authoring and presentation

Interactive multimedia systems can be roughly categorized by what kind of authoring and presentation metaphors they employ. There are three main ways of authoring presentations on a computer—through an ultra-high-level visual authoring environment, through a high-level scripting language, or through a low-level computer program. Each has its advantages

and disadvantages. In addition, each authoring tool has its own model of how the media will be presented. Some systems are card-based (in a manner similar to Hypercard on Macintosh computers), and others are time-based, meaning that there is certain "playout" quality present. Other tools allow the author to combine these paradigms in different ways.

Icon-based visual (non-scripted) systems have the distinct advantage that the author need not type a single line of code to create a presentation. Media objects like images, video, sound, and animation are typically created by using external software, such as paint programs, video digitizers, or word processors, but sometimes simple editors for these objects are provided internally. The authoring environment then provides a set of graphical tools for arranging and controlling the state of these media objects over space and possibly time. The vast majority of these systems are built to deliver presentations on a single video screen or computer monitor.

The problem with visual authoring systems is that they stifle the creativity of the artist by forcing her to work with a set of "mainstream" primitives that the system designers have somehow determined to be most useful. However, it is very often the case that the designers and the artist will differ highly in what is interesting and what is not. It may be possible to create a very good production using the tools given, but it will be hard to accomplish something truly new or breakthrough without extending the system. Many of these systems also force the artist to conform to certain media presentation models, such as the card-based or hyperlink-based metaphor. Further, interactivity in such systems usually only consists of pointing and clicking to "move to the next card" or to cause the playout to take a different course. It would seem that many tool designers believe that the only interesting input device is the computer mouse.

At the other end of the spectrum, it is possible to author a presentation by writing a specialized computer program in a low-level language such as C or C++. The advantage of this route is that expressivity, extensibility, and efficiency are at a maximum. How media objects are presented is limited only by hardware capabilities. Unfortunately, unless the artist is also a computer scientist, the process of learning to program a computer, use

graphics and sound processing libraries, work with operating systems, deal with networks of input and output devices, etc., is likely to frustrate the amateur to an extreme. Of course, an artist could hire several computer programmers and tell them specifically what she wants, but the artist's intentions may not always be accurately expressed. After all, telling someone else how to paint a watercolor or carve a sculpture is not nearly the same as doing it oneself.

Many systems that lie in between these two authoring extremes employ some kind of "scripting language" which in some cases is similar to a programming language, but may provide more intuitive syntax and higher-level constructs useful in multimedia situations. These systems trade off the ease of use of visual authoring systems with the expressivity and extensibility of low-level programs. Very often, visual authoring systems are based on a scripting language level into which the author may enter to add functionality to a presentation, but the author is meant to stay at the visual level most of the time. In other systems, a visual environment is nonexistent and the scripting layer is the primary authoring level.

The function of the scripting language layer is to provide a better interface to libraries of low-level code that actually do all of the work to manage a production. Many of the most successful multimedia authoring systems provide this scripting level to users who want to extend their tools. Not all efforts have been fruitful however. Sometimes writing in the scripting language can be almost as difficult as writing in a lower level language. Other times the language is not a complete programming environment, making it impossible to express certain processes and algorithms. The interface from the language to the low-level processing routines may be very inefficient or restrictive. Perhaps the most important problem is that the vast majority of recent systems handle interactivity and personalization in an unbelievably crude and inelegant manner.

2.2 Macromedia Director

The most popular multimedia authoring system in use today is Macromedia's Director [Mac95]. Director is timeline-based and emphasizes playback ability on personal computers and, more recently, on the world wide web. It allows the author to create "actors" of different media types and then arrange how the presentation will proceed in a score-like visual interface, defining when each actor is on or off the "stage." The latest version claims to support "seamless" cross-platform compatibility (between the only two supported operating systems: Macintosh and Windows). Once a presentation has been completed, it can be distributed and viewed with a smaller player component.

The main interface of Director does not support interactivity—the underlying scripting language Lingo must be used for that purpose, and even then the author can only create restricted and intermittent user interaction through the usual kind of on-screen buttons, icons, and drop-down menus. Lingo scripts can be attached to any actor to describe how to handle various "events" that involve the actor. External devices like compact disk players and video tape recorders can be controlled from Lingo as well, provided that the correct drivers for these devices are installed. Director has become most popular probably because of its pleasing visual interface and the large number of media manipulation tools present, but going beyond the basics and working in Lingo can be difficult. User-defined procedures and loop constructs are not supported, and language expressiveness is limited mostly to handling events and controlling movie and actor settings. It won't be satisfactory for complex or continuous interactivity, nor is it possible to build presentations for any platforms besides Macintosh and Windows.

2.3 Kaleida ScriptX

Kaleida Labs, founded by Apple and IBM in 1992, has developed a platform-independent system called ScriptX for creating interactive multimedia applications [Kal95]. The system

is composed of three parts: a player, a scripting language and class library, and application development and authoring tools. Programmers write a single application in the ScriptX language that is read by the platform-specific media player component. This allows a single presentation to run the same on different platforms. The language itself is a full object-oriented programming environment with a distinct Smalltalk and C++ flavor. While the entire ScriptX system enables one to make very complex and interesting interactive presentations, there are some aspects of the system that leave something to be desired. For one, the ScriptX language has a fairly complicated syntax and is meant to be used by experienced object-oriented programmers. Like Macromedia Director, much of ScriptX's user interface design depends on there being a mouse and windowing system present in which things like scrollbars and radio buttons can be created that users will click on, but in many situations this environment may not be available, especially on experimental media hardware. Just the sheer size and complexity of the ScriptX system may be a disadvantage as well. The on-line information produced by Kaleida boasts a core library containing about 250 classes and approximately 2500 pages of documentation in 5 volumes—enough to scare away any beginner. In the end, it seems that ScriptX suffers from many of the same problems as Director.

2.4 Oracle New Media and Sybase Intermedia

The Oracle Media Objects authoring environment, part of the New Media system developed by Oracle Corporation, uses a card-based presentation model and is specially designed and optimized for interactive television applications [Ora95]. It uses a visual interface for creating high-level application structures and a scripting language called Oracle Media Talk to control presentations. Sybase, Inc., develops a similar system called Gain Momentum that incorporates the Gain Extension Language [Syb95]. Sybase emphasizes the capability for seamless access to large databases of information, appropriate for kiosks and interactive television as well. In both systems, the artist is constrained to work within the confines of one video monitor with the same tiresome event-based point and click style of interaction.

2.5 Java

Sun Microsystems' Java is a general-purpose object-oriented programming language that is very similar to C++ but leaves out some "unnecessary complexities" for the sake of simplicity [Jav95]. Java applications are completely portable across multiple platforms. Programs run inside of a secure environment for protection against viruses and other forms of tampering. This makes Java ideal for creating applications for internet distribution or execution inside Web browsers. It also makes the system much slower than others since programs must be interpreted inside of the Java "virtual machine." Several libraries are provided for managing input and output, connecting to networks, and creating graphical interface components. As such, Java is not a dedicated multimedia authoring environment, nor does it currently offer any innovative user interaction possibilities. The application developer must also have a good understanding of low-level programming techniques. The main value of Java is perhaps its model of architecture-independence at fairly low levels of the system (like user interfaces)—a model which could prove useful in future multimedia work.

2.6 Other commercial products

There are several other systems available that have a more specific focus in their range of target applications. Apple provides an entirely visual authoring system called the Apple Media Tool as part of their Media Kit for creating multimedia presentations on Macintosh computers [App95]. There is also an Apple Media Language available in this package which provides an object-oriented programming environment for adding functionality to the tools in the system. Special Delivery, a product of Interactive Media Corporation, is also a script-free authoring tool useful for creating simple slide-based applications on Macintosh platforms [IMC95]. Paul Mace Software makes Multimedia Grasp for creating stand-alone multimedia applications for display on DOS based personal computers [PM95]. None of these offer functionality or interactivity beyond what is available in the other

systems discussed.

2.7 MADE

Systems currently in research in laboratories around the world surprisingly don't provide much over what these commercial systems offer. For example, the Multimedia Application Development Environment (MADE) under construction by the European Communities ESPRIT III project [HRD94] is focused on designing a portable object-oriented software library based on C++ for creating interactive multimedia presentations on Unix platforms and personal computers. They model applications as consisting of low-level *toolkits* and higher-level *utilities* which are accessed by a scripting layer. They are not developing a scripting language—other research entities are interfacing MADE with languages such as Python [Pyt95] and Tcl [Ous94]. But in the end, like Kaleida's ScriptX, only very experienced computer programmers will be able to make effective use of this system.

2.8 MHEG

The International Organization for Standardization (ISO) is developing MHEG (Multimedia and Hypermedia Information Coding Experts Group), a standard for what they call a "system independent encoding of the structure information used for storing, exchanging, and executing multimedia presentations." [ME95] The major components of the MHEG model are content data, behavior, user interaction, and composition. Oddly, user interaction consists only of two primitives: selection and modification. It seems very early to even be thinking about formulating a standard for multimedia productions when various aspects of the field, especially user interaction, are still in an infant stage. The appropriateness of standardization in this case at all is also questionable. The successful JPEG and MPEG standards for digital image and motion picture compression created by the same organization were all very well-defined scientific problems. However, interactive multimedia exists in a completely different realm from media compression. It would be ridiculous to even think

about creating a standard for something as artistic as drawing pictures or writing novels, so what is the organization's reasoning behind building a standard framework for multimedia art?

Perhaps the single most disturbing aspect of all these scripting environments is the fact that capabilities for interactivity have not progressed past the point and click era. Multimedia may have been reinvented, but user interaction is still stuck in the dark ages. Of course, a mouse and a keyboard are available on almost every computer in existence today, and most of these systems are designed for creating presentations for mass consumption. It can also be argued that the input *devices* themselves are not what matters—what they control in the presentation is what's important. Still, most of these products leave little or no room for experimental input and output devices and models of interaction, and all of them highly discourage or make it impossible to work outside of the restrictive environment of a single video screen.

Chapter 3

The scripting language

The first part of this thesis will be to design a scripting language that will be the basis for an authoring environment for experimental interactive and personalized media that minimizes the disadvantages of the various systems and models discussed earlier. Based on the successes and failures of previous research, many conclusions can be drawn immediately about what features should and should not be present in such a language:

- The scripting layer will be the primary authoring level of the system in order to retain a high level of expressivity. A visual interface could be built at a later time to augment the scripting language layer but will not be part of this thesis.
- The language should favor the artist over the computer scientist whenever possible. Simplicity and elegance of syntax is desired, but not at the expense of expressivity. Unlike Macromedia Director and several of the other high-level authoring tools described previously, the language must be useful for controlling applications in which interaction is continuous and possibly very complex. At the same time, the language should be easy to learn, use, and read. Someone who has never used low-level languages like C or Pascal should be able to understand the basics of the language in a day.

- The language should be fully extensible and user-defined recursive procedures should be supported. All of the things expected from a good programming language should be present, like conditionals and loops.
- The language should support several primitive data types and allow the writer to create her own higher-order types using those primitives as building blocks.
- The language should offer data structures that would be useful in simplifying multimedia applications—structures that would not be normally found in a general purpose language.
- All data storage systems should be optimized for fast access and manipulation.
- Since the language will be used to control live presentations, its design should emphasize efficiency in areas critical to real-time performance.
- The language should be interpretive (as opposed to compiled) to support instantaneous incremental updates and easy modification and debugging.
- The language interpreter should be completely platform-independent.
- The language should be readable and writable by humans and also writable by machines (as the output of a supplemental authoring tool for example).
- There should be a low-overhead mechanism that would allow the script writer to use functions written in lower-level languages in case extra computational efficiency or functionality is desired. This can also be the way the language will connect to libraries that will perform the actual processing to deliver a presentation and control interaction. However, the presence this mechanism should not be an excuse for leaving useful constructs out of the scripting layer.

This list of goals was kept constantly in mind while designing *Isis*, a multi-purpose, high-level interpretive programming language, named after the ancient Egyptian goddess of fertility. Isis can be used as a stand-alone programming environment, or it can function as an interface language to packages of C functions that do specialized operations from networking and file

access to processing graphics and sound. The Isis interpreter itself is written completely in ANSI standard C, allowing it to be ported to any platform that has a standard C compiler. The kinds of data structures available in Isis and its overall design make it suitable for use as an efficient scripting language for interactive multimedia applications. The various features of Isis are explored in the following sections.

3.1 The basics of the interpreter

The *Scheme* language consists of a very simple, intuitive syntax, the basics of which (it is widely claimed) can be learned and understood fairly quickly by people who have never programmed a computer before. For this reason, the syntax of Isis has a similar flavor, but do not be fooled—it is not Scheme. The language core of Isis is small compared to that of Scheme and other languages—providing a manageable set of primitive expressions and data types and allowing the user to create her own higher-order entities from these basics. There are many other internal differences and optimizations that will be discussed later.

The interpreter for this language is a *read-evaluate-print* loop. First, it reads an *expression* that you enter in your terminal (or is read from a file). The interpreter then *evaluates* that *expression* and produces a *value* which is printed on your terminal. Every *value* has a corresponding *type* associated with it, like 'integer' or 'boolean'.

There are 10 primitive types available in the language: integer, real number, character, boolean, procedure, memory address, structure, field, timeline, and type. There are specific ways to express values of each type. For instance, a character constant is expressed as a character between single quotes, like `'q'`. An real number constant is expressed as a number with a decimal point, such as `42.96`.

Isis provides *if-then-else*, *while*, and *switch* constructs for specifying conditional evaluation, and a Scheme-like *let* construct for creating statically-scoped local environments. A rich set of primitive functions for performing mathematics and other operations is also provided.

Appendix A contains a reference manual and tutorial in which more information can be found about the syntax and general use of Isis. The rest of this chapter will be devoted to describing special innovations in the language and its internal design.

3.2 User-defined higher-order types

Isis allows you to define your own higher-order "list" types based on primitive or previously-defined types via the **newtype** construct. A list type is created by specifying a name for the new type and declaring what the types of the elements of the list should be. For example, you may want to create a new type named **Pos** which would be a collection of 3 real numbers. Or you may want to create an **IntList** type that would be a collection of an arbitrary number of integers. Or, instead, you may want to create a **PosList** type that is a collection of any number of **Pos** values. Other even stranger types can be expressed as well.

```
-> (newtype (Pos Real Real Real))
<Null value>
-> (newtype (IntList Int ...))
<Null value>
-> (newtype (PosList Pos ...))
<Null value>
-> (newtype (PlainList Unk ...))
<Null value>
-> (newtype (Strange Int Unk Bool))
<Null value>
-> (newtype (Weirdo Char Bool Int Real ...3))
<Null value>
->
```

The lines above demonstrate how to create different kinds of list types. The ... in a definition means that a list of that type may be of arbitrary length. The **PlainList** type definition uses the special keyword **Unk** to specify that there is no restriction on the types of the elements in that kind of list. The **Weirdo** type uses a special form of ... followed immediately by a number to express an arbitrary-length list in which the last 3 types will be repeated in that order.

Once list types have been defined, they may be used in the same way as any other type. To form a value of a particular list type, the type name followed by the elements of the list is enclosed in parentheses. When a list value is entered, Isis will "sanity check" the values that you provide against the original type definition and report any mismatches—a useful debugging feature to prevent the propagation of errors. Although very inexpensive, this type-checking feature may be turned off once the script is running reliably and extra efficiency is desired.

```
-> (Pos 1.0 2.0 3.0)
( Pos 1.000000 2.000000 3.000000 )
-> (IntList 0 7 2)
( IntList 0 7 2 )
-> (IntList 3 42 -2553 68 20)
( IntList 3 42 -2553 68 20 )
-> (PosList (Pos 3.2 4.3 -6.4) (Pos 1.2 -3.4 5.6) (Pos -4.0 -2.3 4.5))
( PosList ( Pos 3.200000 4.300000 -6.400000 )
          ( Pos 1.200000 -3.400000 5.600000 )
          ( Pos -4.000000 -2.300000 4.500000 ) )
-> (PlainList True 45 's' -89.4)
( PlainList True 45 's' -89.400000 )
-> (Strange -96 'q' False)
( Strange -96 'q' False )
-> (Weirdo 'w' True 10 -3.0 False 99 2.0)
( Weirdo 'w' True 10 -3.000000 False 99 2.000000 )
->

-> (set x 42.0)
42.000000
-> (set y (Pos x (* x 2.0) (* x 42.0)))
( Pos 42.000000 84.000000 1764.000000 )
-> (PlainList x False y)
( PlainList 42.000000 False ( Pos 42.000000 84.000000 1764.000000 ) )
->
```

```
-> (Pos 2.0 'q' 42)
* List type 'Pos', item 1 value type was not Real
* List type 'Pos', item 2 value type was not Real
( Pos 2.000000 'q' 42 )
-> (set-typecheck False)
<Null value>
-> (Pos 2.0 'q' 42)
( Pos 2.000000 'q' 42 )
->
```

Conveniently, higher-order values that contain numeric elements can be processed by the same basic mathematical operations as the primitive types:

```
-> (+ 4 7 9)
20
-> (+ (Pos 1.0 2.0 3.0) (Pos 4.0 6.0 8.0) (Pos -10.0 5.0 -9.0))
( Pos -5.000000 13.000000 2.000000 )
-> (* (IntList 3 4 5 6 7 8) (IntList -2 -3 -4 -5))
( IntList -6 -12 -20 -30 7 8 )
->
```

Strings of characters are modeled by a built-in higher-order type called `String` which contains an arbitrary number of values of type `Char`. A shorthand method of expressing `String` values is provided by Isis to save the work of typing out each individual character constant in the list:

```
-> (String 'o' 'b' 's' 'e' 'q' 'u' 'i' 'o' 'u' 's')
"obsequious"
-> "obsequious"
"obsequious"
->
```

Isis is a dynamically typed language, meaning that you do not have to declare types for variables and procedure arguments. All type checking is done at run time. If the list value type-checker is turned off, Isis will only check types when absolutely necessary—for example, to check if the procedure in a call is really a procedure. Internal primitive operations and C functions linked into Isis also commonly check the types of their arguments and will report errors accordingly.

3.3 Procedures

Unlike other scripting languages, Isis gives you the ability to define your own procedures through the `proc` construct (similar to `lambda` in Scheme). Procedures can be recursive—meaning they can contain calls to themselves. Procedures are then applied by entering the procedure followed by its arguments, all enclosed in parentheses. Here is the definition and an application of the factorial function:

```
-> (set fact (proc (a) (if (<= a 1) 1 (* a (fact (- a 1))))))
< Procedure >
-> (fact 10)
3628800
->
```

In the example above, the variable `fact` is bound to a value of type 'procedure'. Procedures, as well as all other data structures in Isis, are *first-class* data, meaning that they can be passed to and returned from other procedures. Here is a procedure which accepts a procedure and returns new procedure. The new procedure applies that procedure to its argument and adds 42 to the result:

```
-> (set makeadd42
     (proc (theproc)
       (proc (thearg) (+ 42 (theproc thearg)))))
< Procedure >
-> (set factadd42 (makeadd42 fact))
< Procedure >
-> (factadd42 10)
3628842
->
```

If the procedure is not recursive, you do not have to name it by assigning it to a variable in order to use it. For example, the following would suffice:

```
-> ((makeadd42 fact) 10)
3628842
-> (((proc (theproc)
       (proc (thearg) (+ 42 (theproc thearg))))
     fact)
    10)
3628842
->
```

3.4 Interface to external functions and libraries

One of the most powerful features of Isis is that in addition to defining procedures inside the script interpreter, you can also define your own C functions and use them in the interpreter just as you would any other procedure. From the scripting language's point of view, C functions behave in exactly the same manner as scripted procedures. When you apply a procedure, you might be calling a C function or a scripted procedure—and you would never know the difference. In fact, all of the basic mathematical primitives and other operations in Isis are implemented through this interface!

C functions can be written if a highly efficient routine is needed for some part of the program. They can also be used to connect to externally developed C packages, like graphics and sound libraries or user interface software. The structured video and audio packages described in the next chapter make use of this interface.

Efficiency is often a concern in scripting languages that have the ability to link to external functions. For various reasons, fundamental data types in some scripting languages are specially designed at the bit level to be as small as possible. In other systems, such as Tcl, internal script data is stored and processed in string form. The disadvantage of languages of this sort is that the internal representation of data must be converted to and from normal C types in order to link with C functions. This translation is often very costly, especially if it involves converting between strings and numerical types. In Isis, however, all internal script data is stored using standard C data types, thereby minimizing this conversion phase for optimum efficiency when using external C functions.

Defining C functions for use in the interpreter is a fairly simple process. The arguments for the C function are passed in a standard argc/argv format where each argument is a "value" from the script interpreter. Special C macros are provided to manipulate and extract data from this value format and also to build return values. Each function is registered with the script interpreter so that it will become a part of the default top-level environment when the interpreter is started up. Then it can be used inside Isis like any other procedure.

3.5 Structures and fields

Lists are an efficient way of storing *ordered unnamed* pieces of data. However there may be situations where you might like to have something like a C *structure* for storing data. Isis provides that through the `Structure` data type. A `Structure` is an *unordered* collection of *named* fields, each of which contains a value.

When first created, a structure is completely empty—it contains no fields. Fields are then added to an empty structure, each of which has a *reference value* which can be any Isis value. This value acts as a name for the field so that you can retrieve it later. Once a field is obtained from a structure, you can set or query its value. The reason structures and fields are conceptually separate entities is to allow you to directly reference a particular field after you've found it just once, rather than repeatedly search for it inside a structure (a process which could require many value comparisons).

```
-> (set soundstrc (new-structure))
< Structure >

-> (add-field soundstrc 42)
< Field >
-> (add-field soundstrc 'f')
< Field >
-> (add-field soundstrc "pcsition")
< Field >

-> (soundstrc "position")
< Field >

-> (set-field (soundstrc "position") (Pos 3 2 1))
( Pos 3 2 1 )
-> (set-field (soundstrc 42) True)
True
-> (set-field (soundstrc 'f') "perturbation")
"perturbation"

-> ((soundstrc 'f'))
"perturbation"
-> ((soundstrc 42))
True
-> ((soundstrc "position"))
( Pos 3 2 1 )
->
```

3.6 Timelines

In addition to simple lists and structures, there is a third innovative data storage possibility available in Isis called a `Timeline`. When you first create a timeline, it is completely empty. You then place values at any real-numbered point, or time, on the timeline. When you enter each point's value, you also indicate whether you want the value of the timeline to be interpolated between that point and the point directly before it on the timeline. You can specify either linear or cubic interpolation. The timeline therefore has a "value" at every real-numbered point in time which can be retrieved dynamically!

The values in the timeline do not have to be numeric, nor do they have to be of the same

type. However, interpolation will only occur if the key points involved are of the same type, and are numeric in some way. If a list type combines numeric and non-numeric types, only the numeric parts will be interpolated. Timelines are wonderful for expressing time-dependent quantities, but they can also be used to express spatial variances (or variances in any other analogous domain). Explicit linear and cubic interpolation functions are also provided so you can do interpolation outside of timelines if desired.

```
-> (set dave-position (new-timeline))
< Timeline >

-> (key dave-position 0.0 (Pos 2.0 4.0 6.0))
( Pos 2.000000 4.000000 6.000000 )
-> (key dave-position 20.0 (Pos 10.0 42.0 85.0) linear)
( Pos 10.000000 42.000000 85.000000 )
-> (key dave-position 30.0 (Pos -34.0 -40.1 -234.4) cubic)
( Pos -34.000000 -40.100000 -234.400000 )
-> (key dave-position 40.0 (Pos 23.3 354.3 123.4) cubic)
( Pos 23.300000 354.300000 123.400000 )

-> (dave-position 10.0)
( Pos 6.000000 23.000000 45.500000 )
-> (dave-position 4.6793)
( Pos 3.871720 12.890670 24.483235 )
-> (dave-position 20.3)
( Pos 9.369685 41.024039 80.745221 )
-> (dave-position 28.5)
( Pos -32.391944 -52.456888 -219.376075 )
->
```

3.7 Data storage and memory management

Isis employs its own memory management system, thereby avoiding serious inefficiencies in dynamic memory allocation that are present on many platforms and in other systems that use the standard C or C++ memory allocation mechanisms. Isis allocates a certain number of each kind of data structure used in the interpreter and maintains an array of pointers into that block of structures. Highly optimized routines provide constant-time allocation and deallocation of each type of structure for the interpreter. If the supply of one kind

of structure runs out, a new block of structures is allocated and their addresses are added to the pointer array. In this design, memory fragmentation and garbage collection are not issues. Memory statistics functions are available to keep track of how many of each kind of structure are used and the maximum number used at any one time.

Another important optimization feature in Isis is that "lists" are not based on linked lists (as they are in Scheme for example). Lists in Isis are formed with array-based data structures to provide constant time access to any element in the list, greatly improving efficiency in many situations. These arrays hold *pointers* to the elements in the list, not the elements themselves, so the actual elements may be of different sizes. The only problem with this method is that changing the number of elements in a list, through an "append" or "sublist" operation for example, requires creating an array of the new size and copying the element pointers to the new array. For small lists, there is no real degradation in performance compared to a linked-list model, but for larger lists, there will be a slight drop in speed when doing these operations. However, this usually an acceptable tradeoff since in the majority of applications (especially multimedia applications), constant-time reference is much more important than being able to dynamically modify the lengths of large lists.

Chapter 4

Structured audio and video

What is *structured video*? Traditional video is represented in terms of pixels, scanlines, and frames. On the other hand, *structured video*, or *object-based video*, is represented in terms of several component *objects* that are combined in some manner to produce the video scene. These objects may be of many different kinds—two dimensional images or movies, three-dimensional models, text, graphics, etc. They are manipulated and assembled together in different ways over time to generate each frame of a presentation. Representing video in this way tends to simplify greatly the problems of interactive and personalized delivery. Similarly, the term *structured audio* refers to a representation for audio in terms of *objects*, each of which is a piece of audio. Several audio objects are combined in some manner over time to produce a full audio presentation.

Since the Isis scripting language has many useful features, such as the timeline construct and the interface to C routines, packages of Isis functions could be developed to serve as an authoring environment for highly interactive structured audio and video presentations. A list of criteria for the success of this work follows:

- The artist's hands should not be tied by forcing her to work with a particular model of media presentation. Maximum expressivity is the highest priority. The system should be general enough, especially at the levels of interactivity and personalization,

to adapt easily to any kind of application.

- The author should be able to create continuous and highly complex user interaction in a platform independent manner so that a wide variety of input devices could be used. No window manager or graphical user environment should be assumed to be present.

- The system must be fast and support instant updates and on-the-fly modifications. Isis is the perfect selection because of its interpretive nature, its array-based internal structures, and its efficient C procedure call mechanism.

- The core package of Isis functions should be platform-independent so that the same script could be run on any delivery machine. On the other hand, the low-level audio and video processing routines underlying the system should be tailored to take advantage of any special hardware capabilities of each platform, such as parallel processing.

- The system should abstract out a small set of key elements of structured audio and video presentations. Scripts should never contain any low-level platform-specific details.

- Many media types must be supported, including two- and three- dimensional still and animated graphics, text, and multi-track audio. Standard file formats should be readable by the system.

- The system should be able to run in real time on platforms where it is possible (such as the Cheops processing system).

Three packages have been developed for Isis with these aims in mind—one for structured video, the second for structured audio, and the last for user interfaces. There are three conceptual layers to the system (see Figure 4.1). At the highest level is the platform-independent Isis script interpreter. Each package is then accessed in Isis merely by calling various built-in Isis functions. These functions are also machine-independent, so it is possible to write a single script that will run on any platform. It is then the job of each package

to use machine-specific routines and libraries to deliver the presentation as effectively as possible on the given machine.

4.1 Structured Video

The model of the structured video package consists of seven different kinds of abstract entities: objects, engines, actors, cameras, environments, windows, and effects (see Figure 4.2). Essentially, consider yourself as the *camera*, the other people and things in the production as the *actors*, and the details of how you perceive those people and things as the *environment*. The *window* describes how what the camera sees will be translated to a window on a video screen. An arbitrary number of *effects* can be attached to each actor or window to describe special transformations that you want applied to these entities. *Objects* hold the actual graphics data used by actors. The *engine* can loosely be thought of as the *translator* from the virtual world of the presentation to the output device in the real world. Using the information stored in a certain camera, an environment, one or more actors, a window, and all associated effects, the engine does all the work to build frames of a presentation. You can have more than one engine in a presentation, each of which can share actors, cameras, etc., with any of the other engines. This gives you the power to control several presentations in many different windows simultaneously.

A set of built-in Isis functions will allow you to create instances of these seven entities and modify parameters inside each of them to alter their states in some way. Functions are also available for debugging and controlling timers that can be used for synchronization. A typical script will begin by calling initialization functions and creating all the needed entities. To output a each frame of a presentation, the script need only set the appropriate parameters inside each entity to desired values (possibly taking into account timers or user input) and then call a function to actually draw the frame. If this "update and draw" routine is performed repeatedly, it is possible to achieve essentially real-time output!

The following sections describe the seven entities and their parameters in greater detail. For

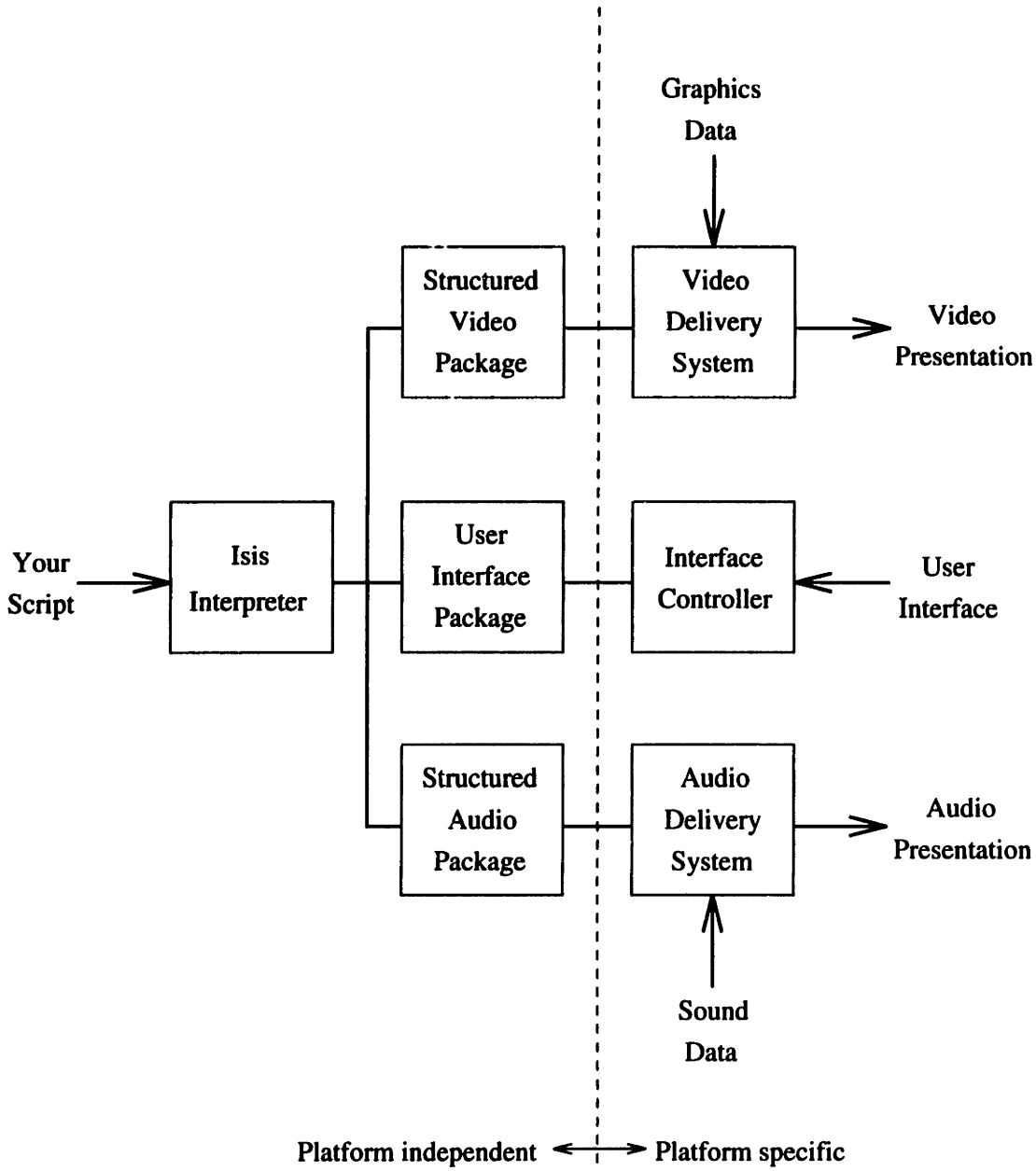


Figure 4.1: *The layout of the Isis interactive structured audio and video system.*

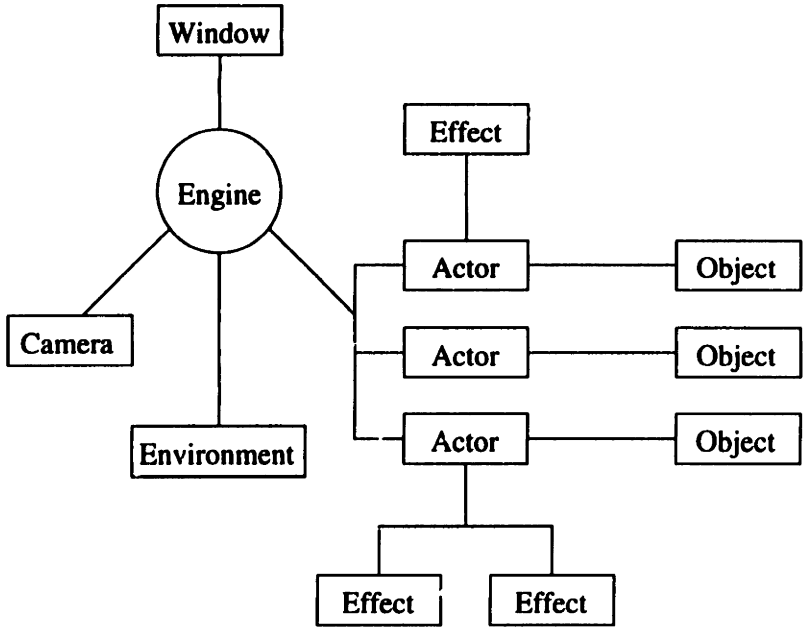


Figure 4.2: *The framework of the structured video package. This is just one possible arrangement of entities.*

more information about the specifics of the system (such as the types and default values of parameters), please see the structured video reference manual in Appendix B.

4.1.1 Objects

Each actor must point to an *object*, which is an entity that refers to actual graphics data, such as an image, a movie, a three-dimensional model, a font, etc. More than one actor can use the same object simultaneously, but each actor can only point to a single object at a time. Only raw graphics data (or information about where it is located) is stored in the *object*. Information about how that data gets rendered is part of the *actor*.

The parameters that can be set in the object are:

- *Source filename*: The filename or URL where the graphics data may be found. URL support may or may not be present depending on the machine. The format of the data is inferred from its filename or extension.

- *Internal name*: Can be anything as it is only used in printing debugging information about the object, not in any processing.

- *Class name*: Used to group certain objects in order to aid the caching mechanism in the delivery system. For example, if there are several objects which are used as backgrounds but only one is ever used at any one time, these objects should be given the same class name, perhaps something like `"background"`. The same should be done for other groups of objects, each group with its own unique class name.

4.1.2 Engines

The top-level entity in any presentation is the *engine*. Parameters inside the engine point to a certain *window*, *camera*, *environment*, and a list of *actors* that should be processed by the engine. There are also parameters for controlling some high-level engine behaviors. Here is the full list:

- *Window*: The window entity to which to render.

- *Environment*: The video environment entity to use in rendering.

- *Camera*: The camera entity to use in rendering.

- *Actors*: Zero or more actor entities for this engine to render.

- *Clearing flag*: Indicates whether or not to clear the output buffer after building each frame. In certain presentations, clearing may not be necessary and can save a little time, or clearing may not be desired for artistic reasons.

- *Depth clearing flag*: Indicates whether or not to clear the output depth buffer after building each frame.

- *Render in order flag*: Indicates whether or not to render the actors in the same order they were specified for this engine. On certain platforms, you may be able to take

advantage of parallelism by switching off this flag, but in other situations, it may be necessary to render actors in a certain order.

- *Output descriptor*: A name for the output of this engine that will be used as a title for the output window if the delivery system supports named windows. It could also be used for other purposes.

4.1.3 Windows

A *window* entity specifies how the view obtained by the virtual *camera* will translate to a window on the output screen. There are 3 parameters the script writer can modify:

- *Location*: The location of the window on a larger screen.
- *Size*: The horizontal and vertical size of the window in pixels.
- *Effects*: A list of special effects entities to be applied to the window (such as brightness or color filters).

4.1.4 Cameras

The *camera* represents the details about a virtual "viewer" in a three dimensional space. The parameters function much like a camera in the computer graphics world:

- *View reference point*
- *View plane normal*
- *View up vector*
- *Eye distance*
- *Viewport*

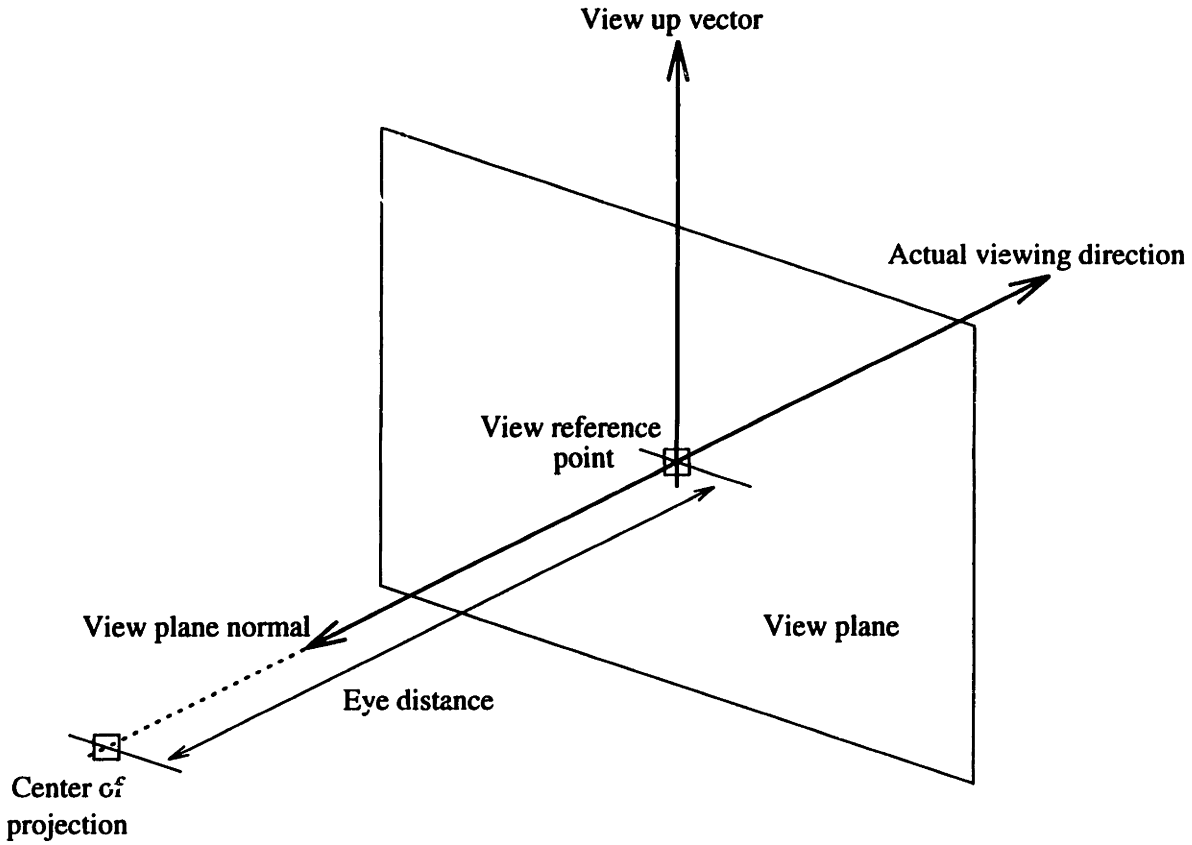


Figure 4.3: *Elements of the virtual camera.*

The *view reference point* and *view plane normal* determine a plane in space called the *view plane.* The *center of projection* is located *eye distance* units away from the view reference point in the direction of the view plane normal. (Note that the view plane normal points *toward* the viewer—see Figure 4.3.)

An arbitrary point in the three-dimensional space is projected onto the view plane by finding the intersection of the view plane with the line that passes through both the center of projection and that point. The entire three-dimensional scene is mapped onto the two-dimensional view plane in this manner.

The portion of this view plane that is visible in the output window is called the *viewport.* You first specify the direction to be considered as "up" as the *view up vector.* This vector points in the positive y direction on the view plane. Then, the viewport is specified by

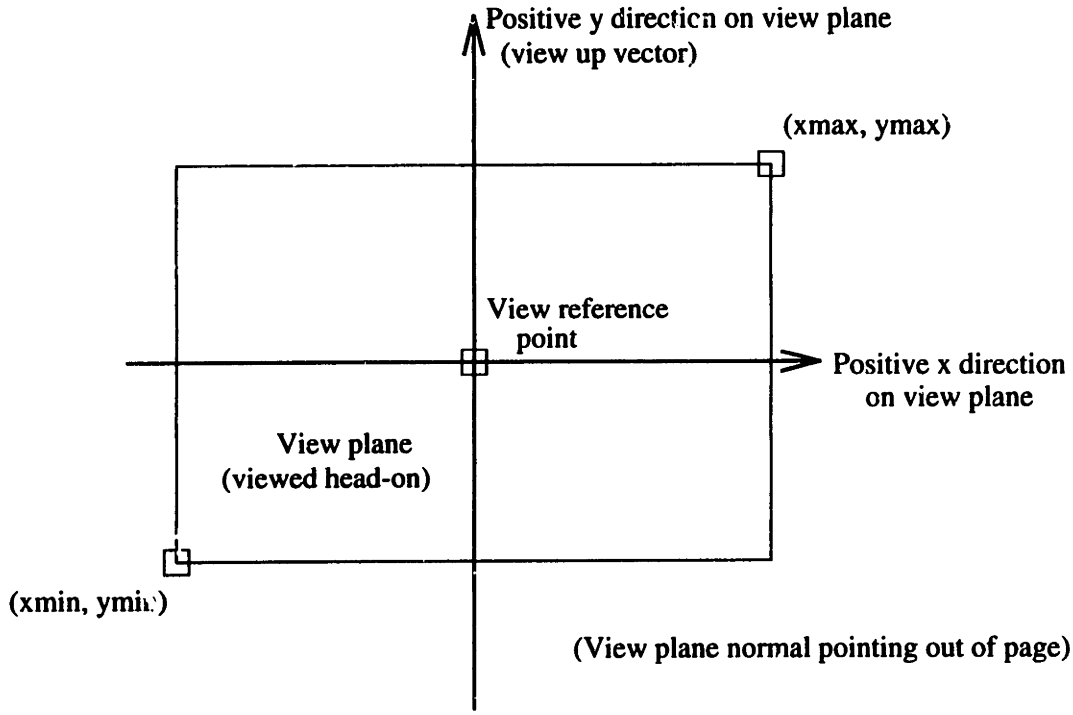


Figure 4.4: *Elements of the viewport.*

considering the view reference point to be the origin on the view plane and then giving a minimum and maximum x and y extent for the sides of the viewport (see Figure 4.4).

All of the distances and dimensions needed for the camera parameters are specified in *world units*. Whether the world units are millimeters or miles depends on the units of the *objects* in the presentation. If the objects are three-dimensional models, the same units that were used when those objects were originally created are the units that should be used for the camera. If the objects are two-dimensional images or movies, some file formats will allow the creator to specify a pixel-to-world scale factor for the images which will indicate the proper units to use. If no scale factor is specified, then the pixel size is assumed to be the same as the world size (a 1:1 pixel-to-world scale factor).

4.1.5 Environments

The video *environment* would contain information about how the actors in the scene are perceived by the camera. For instance, it would store information about the location and intensity of lights in a scene. There is a similar "environment" entity in the structured audio package which holds information about the acoustics of the three-dimensional audio space. In the current version of the structured video package, support for lights and other environment factors is not present.

4.1.6 Actors

All of the other people and things in a production are the *actors*. Each actor has these parameters available to be modified:

- *Object*: The video object where the graphics data for the actor resides.
- *Frame*: The temporal frame number of the object to use (if more than one is available).
- *View*: A spatial "view" number of the object to use (if more than one is available).
- *Visibility*: Opaqueness of the actor.
- *Positioning mode*: A flag controlling the positioning mode of the actor. If the actor is in 3D mode, you can position and rotate the actor anywhere in the three-dimensional virtual world, and the actor will be scaled and rotated properly depending on the current camera pose when the frame is plotted. If the actor is the simpler 2D positioning mode, you can only place the actor at a certain pixel position in the current window—the current camera and environment are ignored.
- *Position*: The position of the object's *hotpoint*. By default, this point is the center of the object, but some file formats allow for user-specified hotpoints.
- *Rotation*: The rotation vector for the actor. Rotation occurs about the hotpoint of the object.

- *Scale*: A scale multiplier for the actor's units.
- *Depth flag*: Indicates whether or not to use the depth buffer of the actor's object if it is available.
- *Z-buffer flag*: Determines if the actor will be Z-buffered into the output window (as opposed to simply overwriting the window with the rendered actor).
- *Alpha flag*: Indicates whether or not to use the alpha buffer of the actor's object if it is available.
- *Blending flag*: Determines whether to perform high-quality alpha blending when transferring the actor to the output window (as opposed to treating the alpha channel as if it were binary).
- *Effects*: A list of special effects entities to be applied to the actor before compositing it in the output frame.

4.1.7 Effects

Effects are entities that describe a wide variety of special transformations that could be applied locally (to the graphics data in one particular actor) or globally (to the entire contents of a window)—just like "special effects" in the film industry. Some examples of special effects transformations are: filtering (brightness, color, sharpening, blurring), remapping (scaling, rotating, warping), edge sensation, line detection, pixelation, anti-aliasing, noise addition, noise removal—the list is almost endless. There are separate parameters available to modify for each type of effect. The goal is to embody as many highly-configurable "primitive" special effects in the package as possible, the combination of which could produce just about any effect that is desired. Not all machines may be able to apply all the primitive special effects—each specific platform would have to do the best it can with the routines available.

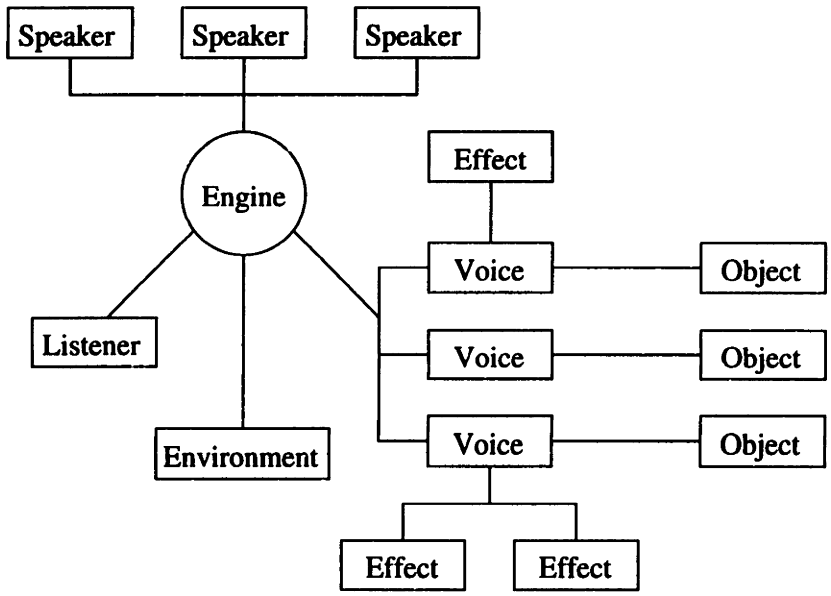


Figure 4.5: *The framework of the structured audio package. This is just one possible arrangement of entities.*

4.2 Structured Audio

The structured audio package lets you create a three-dimensional environment for the playback of sounds. You can place any number of sounds and a single listener anywhere in a virtual space and modify playback and acoustical characteristics in order to achieve a desired effect. The results are rendered on a set of speakers surrounding the real listener in the physical area where the presentation will take place.

The model of the structured audio package is analogous to that of the structured video package, consisting of seven kinds of abstract entities: objects, engines, voices, listeners, environments, speakers, and effects (see Figure 4.5).

As with structured video, consider yourself as the *listener*, the other people and things that make noise in the production as the *voices*, and the details of how you perceive those people and things as the *environment*. The *speakers* describe how what the listener hears will be translated to the speakers in your room. Any number of *effects* can be attached to each voice or speaker to indicate special filtering or other transformations that you want applied

to these entities. *Objects* hold the actual audio data played by voices. The *engine* is the *translator* from the virtual world of the presentation to the output devices in the real world. Using the information stored in a certain listener, an environment, one or more voices, one or more speakers, and all associated effects, the engine does all the work to build the mixed audio for presentation. You can create more than one engine in a single session, each of which can share voices, listeners, etc., with any of the other engines. This gives you the power to control several presentations using many different sets of speakers simultaneously.

There is a set of Isis functions which allows you to create instances of these seven entities and modify parameters inside each to alter their states. Debugging and timing functions are also available, just as in the structured video package. Scripts begin by initializing and creating all the needed entities. In order to actually output audio, the script needs to set the appropriate parameters to desired values (taking into account timers and user interaction) and call a function to render the audio with those settings. The audio plays continuously, but if this "update and render" routine is performed repeatedly, it is possible to achieve essentially real-time feedback for changes.

The sections that follow elaborate on the seven audio entities and the parameters available in each. Appendix C contains a reference manual for the structured audio system in which all the specifics may be found.

4.2.1 Objects

Each voice must point to an *object*, which is an entity that refers to actual audio data, such as a sound effect or a piece of music. More than one voice can use the same object simultaneously, but each voice can only point to a single object at a time. Only raw audio data (or information about where it is located) is stored in the *object*. Information about how that data gets rendered is part of the *voice*.

There are only two parameters in an object which you may set:

- *Source filename*: The filename or URL where the audio data may be found. URL support may or may not be present depending on the machine. The format of the data is inferred from its filename or extension.
- *Internal name*: Can be anything as it is only used in printing debugging information about the object, not in any processing.

4.2.2 Engines

Like the structured video package, the top-level entity in an audio presentation is the *engine*. The parameters are:

- *Speakers*: The speakers to which to render audio.
- *Environment*: The audio environment entity to use in processing.
- *Listener*: The listener entity to use in processing.
- *Voices*: Zero or more voice entities for this engine to render.
- *Output descriptor*: A name for the output of this engine.

4.2.3 Speakers

A *speaker* entity specifies where the physical speakers are placed with respect to the listener in your presentation room. This information is needed in order to render the correct directional mix of the audio for each speaker. There are only two parameters you can modify:

- *Effects*: A list of audio effects entities to be applied to the output of the speaker.

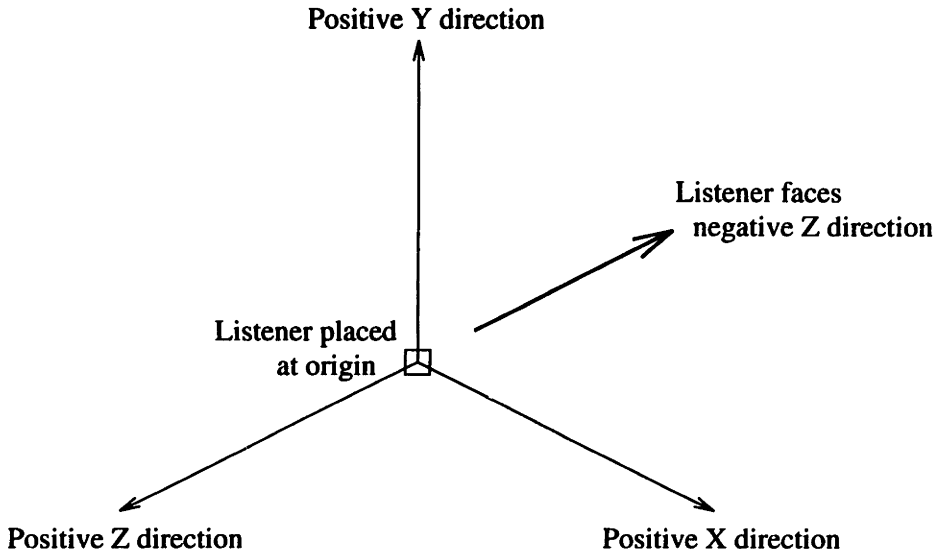


Figure 4.6: *The coordinate system of the structured audio physical presentation space.*

- *Position*: The position of the speaker in the physical presentation space, given in a coordinate system where the physical listener is at the origin and is facing in the negative Z direction. The positive X axis points to the right of the listener, the positive Y axis points upward, and the positive Z axis points backwards (see Figure 4.6).

4.2.4 Listeners

The *listener* represents the details about a virtual "hearing" entity in a three dimensional space. The parameters function much like an extremely simple camera in the computer graphics world:

- *Position*: Position of the virtual listener in a rectangular solid "room" with its origin at the exact center. (The dimensions of the room are specified in the *environment* entity).
- *Normal vector*: Vector pointing the *opposite* direction that the listener is facing.
- *Up vector*: Vector pointing in the direction considered "up."

4.2.5 Environments

The audio *environment* contains information about how the voices in the scene will be perceived by the listener. For instance, the environment stores information about the geometry and acoustical properties of the virtual "room" where the actior is taking place. In the current version of the package, this room is considered to be a rectangular solid in shape. Here are the parameters:

- *Dimensions*: The dimensions of the rectangular solid room.
- *Reflection coefficients*: Coefficients for reflection for each of the six walls of the room.
- *Maximum number of reflections*: The maximum number of virtual "bounces" a sound can make in the room. The listener will only hear virtual sound sources that have reflected off the walls less than this many times. Setting this number high increases the realism of the audio reproduction and, consequently, the amount of processing required.
- *Reverberation constant*: Amount of reverb to apply to sounds in the room.

4.2.6 Voices

All of the noises emitted by the people and things in your production are modeled by the *voices*. Each voice has these parameters available to be modified:

- *Object*: The audio object where the sound data for this actor resides.
- *Entry time*: The place within the sound object to begin playing from.
- *Start time*: Specifies the exact time to begin playing the sound on this voice. The current time can be queried or set using other functions in the package.
- *End time*: Specifies the exact time to stop playing the sound.

- *Position*: The position of the voice in the virtual "room" as described before.
- *Maximum depth*: A distance above which virtual sources of this voice will not be heard—useful for turning off processing for a voice when it is beyond a certain distance.
- *Gain*: Gain to apply to the sound.
- *Loop*: A flag indicating whether or not to loop the sound back to the start if it plays past its end.
- *Effects*: A list of special effects entities to be applied to this voice before mixing it into the composite output.

4.2.7 Effects

Audio effects work just like video effects—they describe a wide variety of possible transformations that could be applied locally (to a particular voice) or globally (to the entire composite audio mix for a particular speaker or engine). Examples of audio effects which might be applied are: high-pass or low-pass filtering, noise addition or removal, pitch adjustment, etc. Each has its own parameters which could be modified to configure the effect. Not all machines would be able to apply all possible effects—each platform would create the presentation to the best of its ability.

4.3 User Interfaces

The structured audio and video packages for Isis allow one to author complex presentations that incorporate a wide variety of media forms. However, presentations of this kind in many cases could be much more compelling if they allowed the viewer or listener to interact in some way. Some kind of interface into Isis would be needed to provide input that could be used to alter the media delivery.

The user interface package for Isis enables the creation of a standard user interface that will

work on any delivery platform, from common workstations and operating systems to experimental audio and video processing hardware. The problem with designing a standard user interface system is that not all machines will offer the same input possibilities as the others. Some systems may manage a windowing system that supports things like buttons and icons, but many times, especially with experimental machines, none of these conveniences are present, and different and sometimes odd input devices must be used instead.

The Isis user interface package solves this problem by making the least number of assumptions possible about the specific capabilities of various platforms. Windows, buttons, scrollbars, menus, etc., are all (happily) assumed to be nonexistent. Event processing also is assumed not to be present. The operation of the package is governed by the simplest possible model of interaction.

The user interface package lets you create and manipulate two different kinds of entities: *interfaces* and *inputs*. An *interface* is nothing more than a collection of related *inputs*. An *input* is an abstract notion of a polled "input device" which can receive data of only 5 different kinds: boolean, integer number, real number, character, or string. Each input device can be configured in different ways to limit the possible range of acquired values.

Each *input* that is created and configured in a script is tied to an input device on the specific platform on which Isis is running. How that input device is realized depends on what input possibilities are available on that platform. For example, an integer input configured to have a certain maximum and minimum value might be rendered as a scroll bar inside some kind of graphical user interface, or, on more primitive systems, it might be tied to a certain knob in a physical knob box connected to the computer.

A typical script requiring a user interface will begin by calling initialization functions and creating and configuring all the needed interfaces and inputs. Then, to obtain input values, the script need only call a function that checks the value of a particular *input* entity. Appendix D contains a reference manual for the user interface package.

4.4 Implementation on the Cheops processor

Currently, Isis and the three packages described in the previous sections are operational on the MIT Media Laboratory's Cheops image processing system, a data-flow computer optimized for real-time processing and display of video [BW95]. In Cheops, several basic, computationally intensive operations (such as image filtering, spatial remapping, motion estimation, etc.) are embodied in specialized hardware. Cheops' high-bandwidth transfer capability (up to 120MB per second), dynamic parallelization of processing [She92], and large memory store (up to 4 gigabytes) make it the perfect platform for real-time interactive structured audio and video experiments.

The Isis structured video package builds upon elements of previous prototype structured video systems on Cheops [Gra95] [Cha95] [BGW94]. Since Cheops is still an evolving machine, certain aspects of the complete package are not yet present. Only two-dimensional objects may be loaded into the system due to the fact that Cheops is not currently optimized for rendering three-dimensional media types. For the same reason, environment factors, such as virtual lights, are also not supported. Objects may, however, contain an alpha or depth channel, and z-buffer processing is enabled. This allows the author to realistically simulate three-dimensionality using only two-dimensional building blocks. Since there is a very limited number of primitive graphics transformations available in hardware at this time, support for *effects* entities is also temporarily unavailable.

The structured audio package on Cheops connects to the host Alpha workstation and uses a special sound server designed to work with the AudioFile system [Ing95]. This server supports all the aspects of the ideal package described previously except for *effects* entities which are currently unimplemented. Only raw 44.1 KHz sound files can be loaded and a maximum of four sounds may be played simultaneously.

The Cheops user interface package connects to an external device containing eight knobs. Each knob can be configured with a certain maximum, minimum, and step value. All input types are supported except for strings.

The entire system on Cheops has been functioning exceptionally for several months and has been used in building several innovative applications. For reference, Appendix E contains a detailed summary of the features and oddities of the Cheops versions of all three packages.

4.5 Applications

In order to determine the success or failure of this part of the work, a broad range of applications have been produced using Isis and the structured audio and video packages. A system for personalizing weather information was created [Eva96] in which the user selects several cities and regions in the United States for which she is interested in knowing the current and forecast weather. An external program then connects to weather information centers on the internet, retrieves the desired information, and outputs an Isis script that can be executed on Cheops to display the personalized weather report. Playout consists of animated maps and graphics to visualize the data, and the viewer can move immediately to certain kinds of information for certain times and places.

The system has also been used to author a narrative that uses story perspective as a model for interaction. This production incorporates continuous interactivity with aspects of play-out personalization to optimally convey a complex story world to the experiencer. Three-dimensional and two-dimensional graphics objects and several audio streams are mixed in real time. This production is explained in greater detail in the next chapter.

Several other applications have been developed, including a glorified pong-style video game that incorporates elements of the interactive narrative described above. Many video-only and audio-only projects have also been completed to test various aspects of the system.

The fact that it has been possible to fully implement such a wide variety of applications demonstrates the system's power of adapting to many different uses. In many ways, its design was partially driven by the wide-ranging needs of the applications that were to be developed. The user interface package is general enough to support most any kind

of interactivity, whether it be passive or active, continuous or intermittent, or a complex combination of these.

The design of the structured audio package allows the author to simulate very simple playback conditions or create complicated three-dimensional audio environments with a minimum of effort. The author need not be an expert in acoustics to achieve a desired effect. Although not all its features have been implemented on Cheops, the structured video package also allows for a high degree of expressivity. Many different kinds of media objects can be manipulated in either a two- or three-dimensional presentation environment. The current implementation still lacks in some areas. A larger library of visual transformations is needed in order to support transparency, blending, and feedback loops. In the future, lighting, shading, and texturing will need to be supported as well. Otherwise, the system has been quite adequate for applications ranging from data visualization to interactive storytelling to arcade-style games.

The Isis language itself has been the perfect base on which to build the interactive media packages described in this chapter. Since it is a complete programming language, it is possible to express virtually any algorithm or process that might be needed. The *timeline* storage mechanism is an extremely valuable tool for expressing time- and space-variant data. Its general simplicity of syntax allows most anyone, especially those who are not computer scientists, to learn and comprehend the basics of the language in a reduced amount of time compared to other multimedia scripting languages. No understanding of object-oriented or event-based programming is necessary. The platform-independence of the interpreter and function packages assures that porting the system to other architectures will be fairly easy.

Chapter 5

An interactive narrative

Computers have long been used extensively for information storage and retrieval, as well as for entertainment. Advances in technology for dynamically mixing different kinds of media were applied initially to these areas to the point where today we can create beautiful informational kiosks and incredible action video games. But it is only recently that some non-computer-scientists have realized that this technology also holds great potential in the creation of interactive pieces of art. Much attention has been devoted to the specific area of storytelling, unfortunately often with less than meaningful results.

Take, for example, some recent experiments with so-called "interactive movies" which involved attaching a small joystick with three buttons to every seat in a movie theater. Audience members were presented with at most three choices at certain key points in a movie that allowed them to collectively decide which direction the plot would take in the story. It is easy to see why experiments like this were dismal failures. Of course, some members of the audience would feel frustrated because their choices could be overruled by the majority, and, in these early trials, less than meaningful stories were delivered by less than skilled actors. But even if the best actors in the world were acting out the most amazing story ever conceived, there are still much deeper problems with this and similar models of interaction.

People watch films and television shows, go to theatrical plays, read books, etc., to experi-

ence and learn from and be emotionally moved by a story. In most traditional productions, these elements are invoked precisely by experiencing a specific line of events and actions occurring in a certain environment with certain kinds of characters. All of these intrinsic components of a story are referred to as the *narrative*. Allowing the experiencer to control aspects of the narrative greatly alters (and usually detracts from) any meaning or impact that might have been part of the original story idea. On the other hand, the *narration* refers to *how* the story is told or presented, involving things like the way actors deliver their lines, the design of the scenery and lighting, the editing style, the musical score, etc. These presentation elements set up the entire mood and atmosphere for a story, allowing its creator to emphasize or deemphasize certain things and to maximize its impact. Allowing the experiencer to control the narration is allowing the entire feel of the story world to change or possibly be compromised.

Considering these simple observations, it seems clear that any successful venture in interactive storytelling is going to have to break completely from the rules of traditional forms and enter a universe of its own, rather than slap gimmicks or bells and whistles onto old ideas. Very few artists have been able to go beyond the horizon to this new realm and create compelling content.

With this in mind, the Interactive Cinema and Television of Tomorrow groups at the MIT Media Lab set out to break from past lines of research and create a completely new kind of interactive story experience. We decided early that we wanted to achieve certain basic goals:

- The story should be short and simple, yet also gripping and meaningful. Shortness is desired for two reasons—because there is only a limited amount of memory available on delivery platforms that can be used to store the media we intend to use, and because in many cases there is a certain feeling of beauty that accompanies simplicity.
- The *narrative* in the production will be held constant. No modification of the actual story will be allowed. The interaction will consist of changing aspects of the *narration*

of the story in order to present a different subjective point of view on the same events.

- For simplicity, the viewer should take a third person perspective, not the part of an actual character in the story. She should also have the ability to move around spatially in the virtual scene.

- Interactivity should be continuous and response to it should be immediate, yet the presentation also should be able to play out in a meaningful manner if no interaction is given.

- The user interface should be as simple and intuitive as possible and provide constraints and visual feedback so that the viewer never gets "lost."

- The playout of the production should not be governed merely by separate pre-ordained sequences of shots and sounds. There should enough parameters, many of which might be interpolated, to make the story space large and provide for a virtually infinite number of possible playouts.

- The experience should be one-on-one in nature. Multiple-viewer theater-like situations should be avoided. At the same time, the production should never feel like or have the goal-oriented nature of a game, beyond simply invoking a desire to explore and understand the story world.

- The production should incorporate aspects of personalization to slightly alter the delivery based on viewing conditions or information about the viewer.

5.1 The story

The story for our production is based on the student film *The Yellow Wallpaper* created by Tony Romain at Boston University in 1995. This film, in turn, is based on the short story by the same name written by Charlotte Perkins Gilman. Scenes of a present-day woman writing this story are interwoven with the world of the character she is creating in the past. The film revolves around her struggle to touch and understand this parallel world

in the past while at the same time dealing with an abusive husband in the present who is completely oblivious to what is happening inside her mind.

This story was perfect for our purposes because of its simple yet powerful conflict between the woman and the man, offering two extremely opposing points of view on the same world. Since brevity was an important issue in our production, we condensed the narrative down to a single very dramatic moment that conveys all the emotions of the entire story. In our short version, the woman (Kathy) is sitting at a desk in an empty attic in her house, hammering away at a typewriter. Her husband (John) arrives home downstairs and we hear his harsh voice penetrate the room, calling angrily to his wife. She does not answer. John walks in the room and proceeds to make several abusive comments. She ignores him—not purposely, but because she is so engrossed in her inner world that she barely recognizes his existence. He walks over to a woman's dress form in the corner of the attic, grabs a dirty rag that is sitting on top of it, and attempts to get her attention once more. Unsuccessful and frustrated, he throws the rag to the floor and storms out of the room, leaving her alone again. Kathy perceives these events, but only on the edge of her consciousness and in a much different light. She is more entranced by images of the story she is creating. For example, instead of grabbing a rag off of the mannequin, she sees John grabbing a handkerchief from the hand of the character she is writing about.

The purpose of making this story interactive was to explore issues concerning the nature of reality and perception. Our goal was not to allow the experiencer to change the actual *narrative*—instead, the main interaction in the production is based on changing aspects of the *narration* in order to convey a different subjective point of view on the same events. This interaction is motivated by the experiencer's desire to enter the minds of the characters and explore their differing perceptions of the same world. By modeling these two points of view as two *extremes* in a "point of view" space, it is also possible to interpolate parameters and generate versions of the playout that *blend* each point of view to different extents. As a secondary point of interaction, the viewer may also change her spatial position and orientation in the virtual attic where the action takes place, but only along a certain constrained path in order to always keep the viewer pointed generally toward the action.

The production is also personalized to provide slightly altered editing and shot composition depending on the size of the display window.

5.2 The scene

Knowing that the Cheops processor was going to be the primary delivery platform for the final presentation, it was necessary to obtain the media objects in a very specific way in order to optimize its performance. Cheops does not have the ability to render three-dimensional textured objects in real time. Therefore, since it is almost impossible to generate an accurate and realistic model of the structure of a human being, let alone its movements and gestures, we knew we would have to shoot real live actors.

Where and *how* we would shoot the actors was another issue. Since we wanted to give the viewer the ability to roam around the attic, at least to a certain restricted degree, it would be necessary to shoot the action from several different angles. To maximize realism, we could have shot the action in a real attic with several cameras, but that would restrict the viewer's motion to those angles only, of which there might very few. It would also hamper our ability to apply transformations or effects to specific elements of the scene. Therefore, we opted for the structured video approach in which a three-dimensional model of an attic would be created that could be rendered from any viewpoint, giving us the ability to synthesize camera angles. The actors would be then shot against a blue screen from several calibrated cameras angles simultaneously and then scaled and composited into the appropriate location in the rendered scene in real time during playout.

In order for this approach to be successful, some kind of realistic representation for the attic needed to be captured. Utilizing recent results in the generation of three-dimensional models from uncalibrated two-dimensional camera views, we were able to create a complete model of the attic with high accuracy. This process is described in detail in [BeB95]. Any number of ordinary photographs of a scene can be used as the raw data for this process. We used eight photographs of a room at the actual house where the film version of *The Yellow*

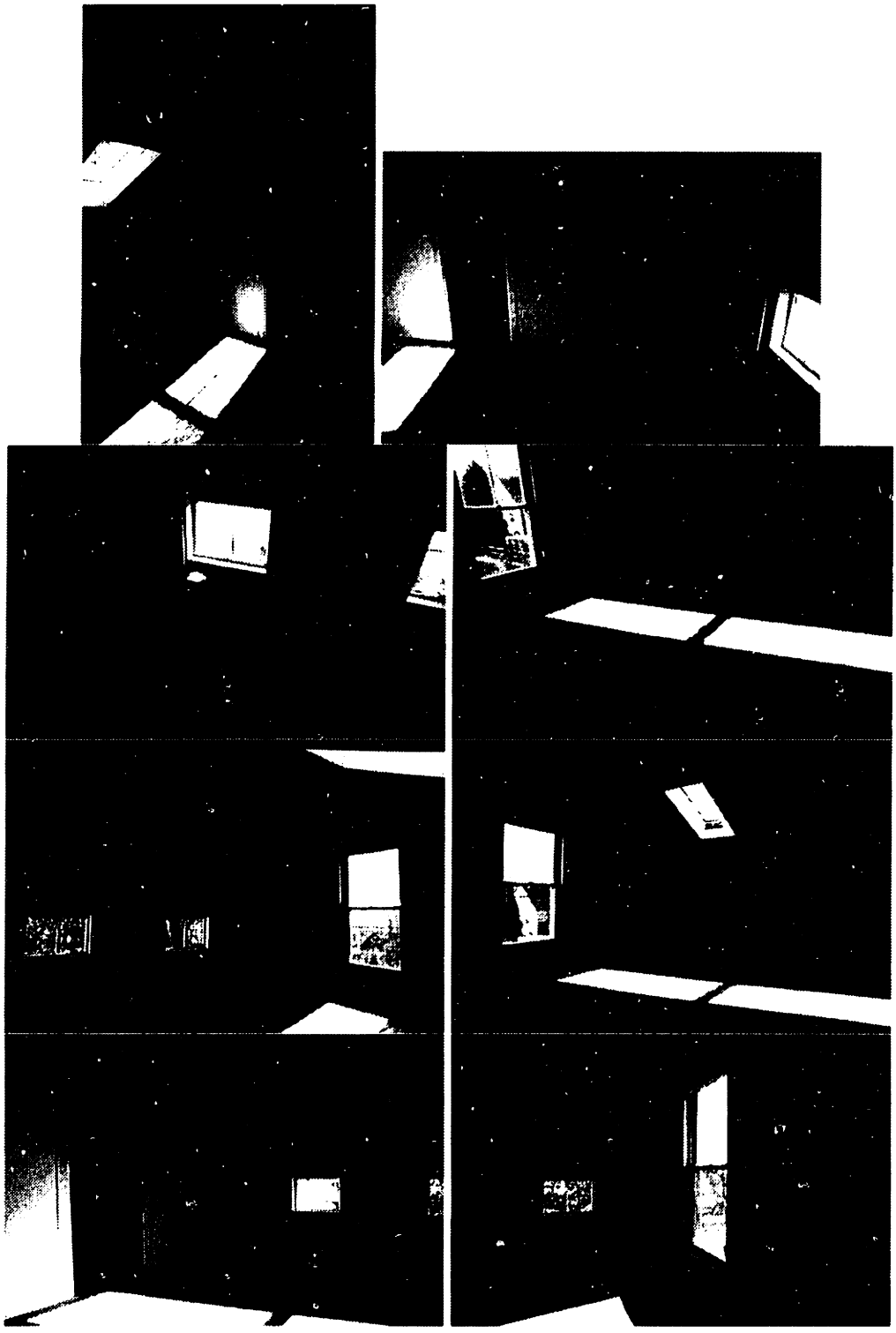


Figure 5.1: *The original eight photographs used to generate the three-dimensional model of the attic.*

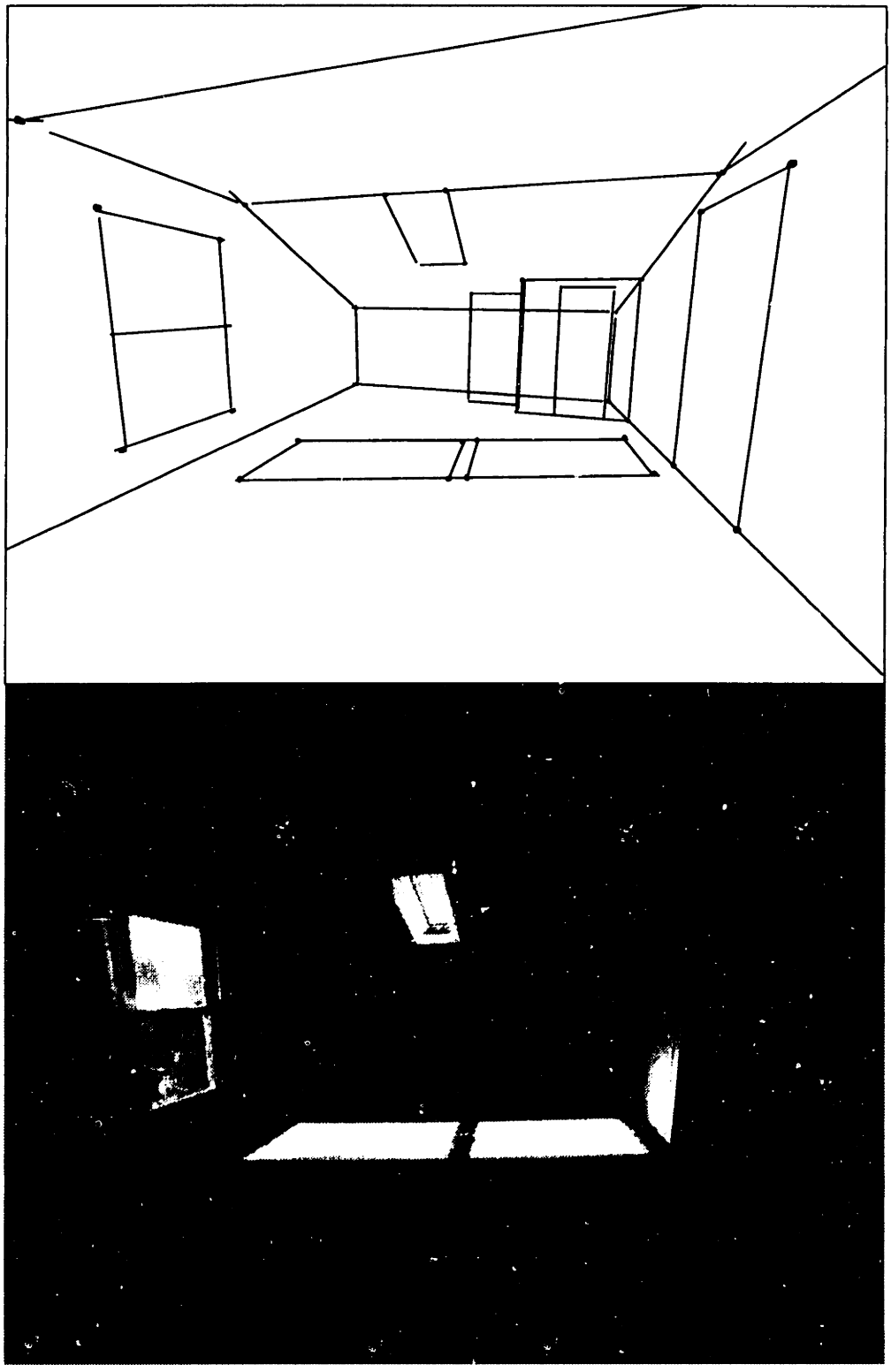


Figure 5.2: *Wireframe and textured images of the attic model.*

Wallpaper was produced (see Figure 5.1).

By detecting long lines in each image and finding the vanishing points of sets of parallel lines in perspective, it is possible to calibrate the cameras with which the original photographs were taken and map the image information onto planar polygonal surfaces in three dimensions. By combining the information from several camera angles, a full textured model of the scene can be produced. Areas of the scene that were visible in more than one photograph can be blended together virtually seamlessly and can enhance resolution in those areas. One rendering of the final three-dimensional model of our attic is shown in Figure 5.2.

5.3 The shoot

Once the model of the attic was finished, the next job was to shoot the actors against a blue screen as if they were performing inside of this virtual set. To facilitate viewing the action from different viewpoints, we shot the scene simultaneously from several angles. In our previous production, *The Museum*, we only used three cameras mounted at eye level. These angles did provide fair coverage of about one hemisphere of the scene, but they did not give us the opportunity to move the virtual camera up or down for more dramatic effects. Therefore, we chose a five camera set-up for the *Wallpaper* production—three at eye level and two mounted at higher angles. Shooting was done with Betacam SP cameras, all of which were connected to a single time code generator to allow for simultaneous recording and to aid in synchronization during post-production.

Specially-marked cubes were placed on the set once final camera positions were determined (see Figure 5.5). These cubes enabled the calibration of the five studio cameras in order to accurately project the positions of the actors into the virtual attic as they walked around the stage.

Each actor was filmed separately from the rest to make the chroma-key segmentation process easier. If the actors were shot together, they would have to be separated not only from the

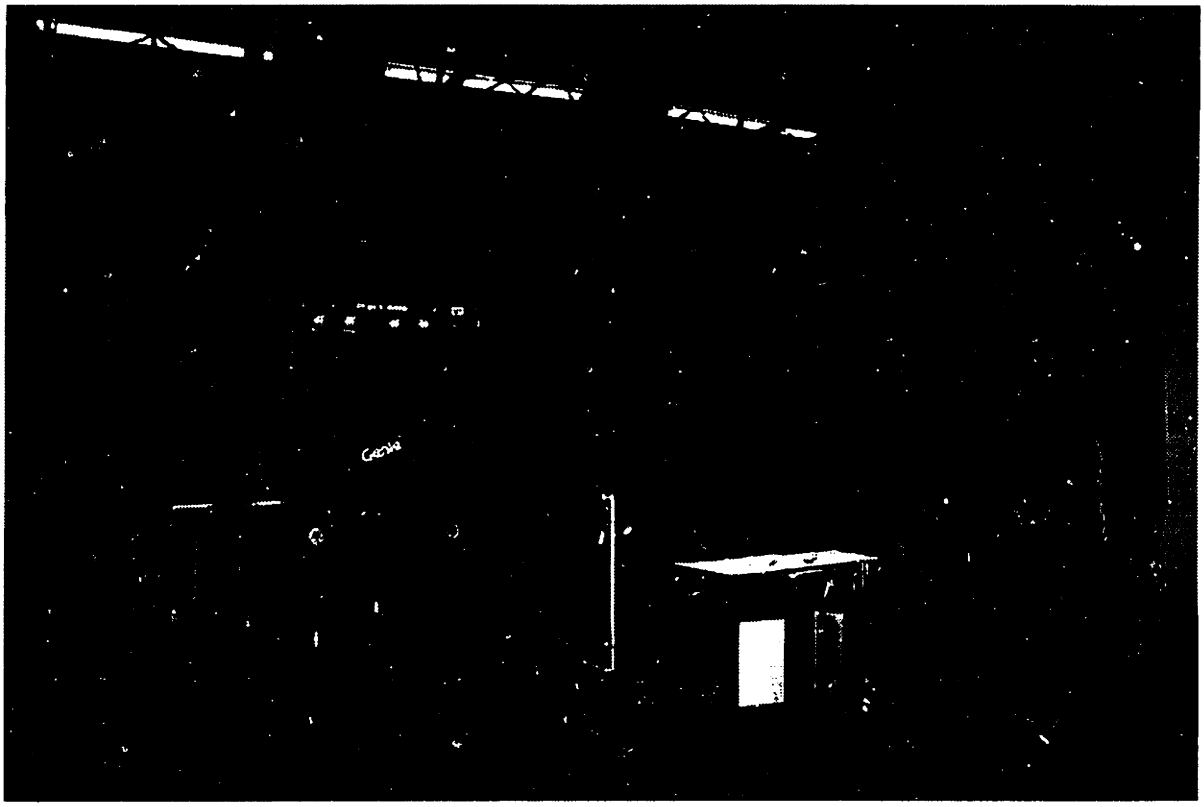


Figure 5.3: *The completely empty blue-screen studio.*

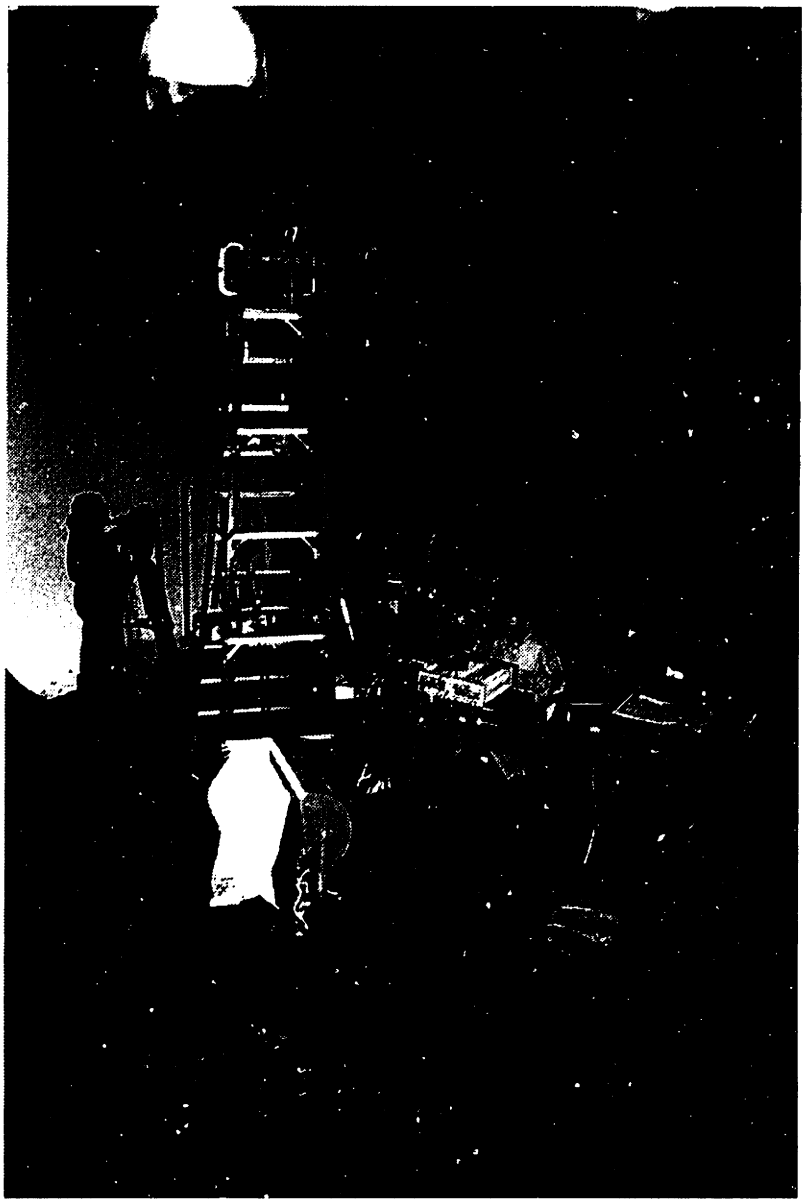


Figure 5.4: *Cameras mounted high and low.*

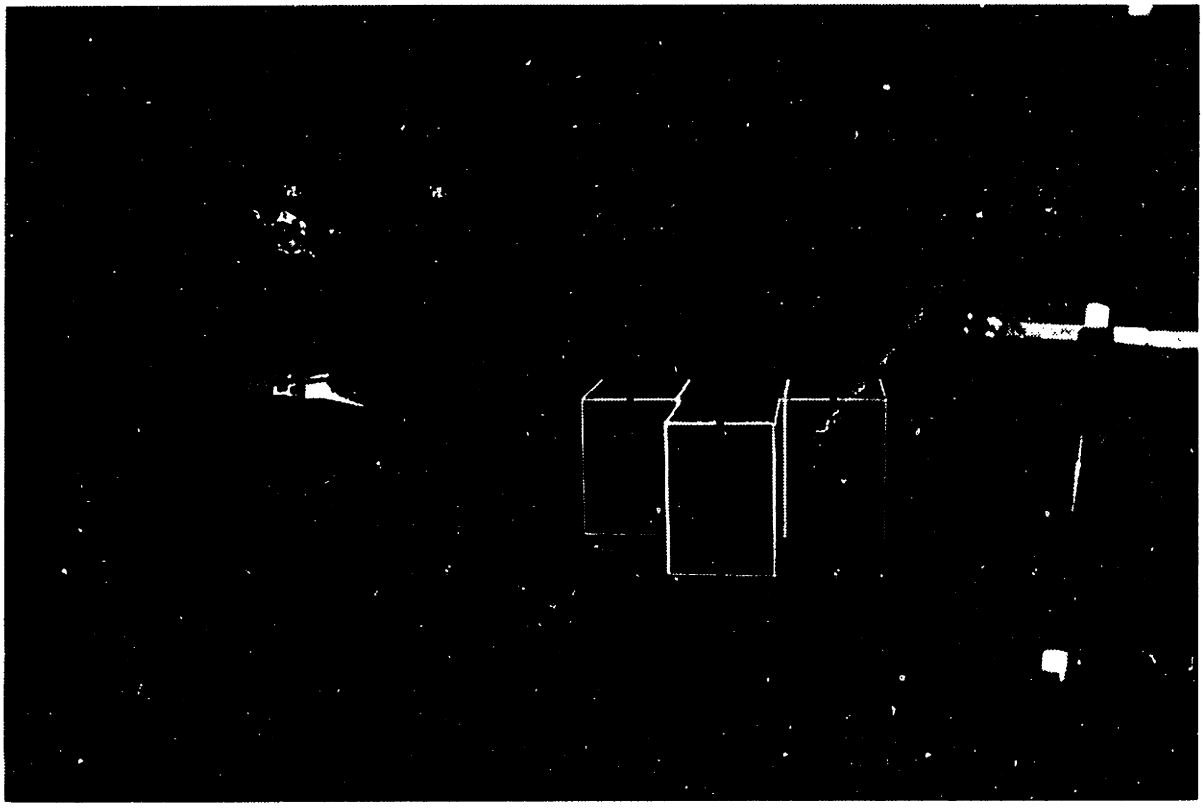


Figure 5.5: *Specially marked cubes are placed on the set for camera calibration.*

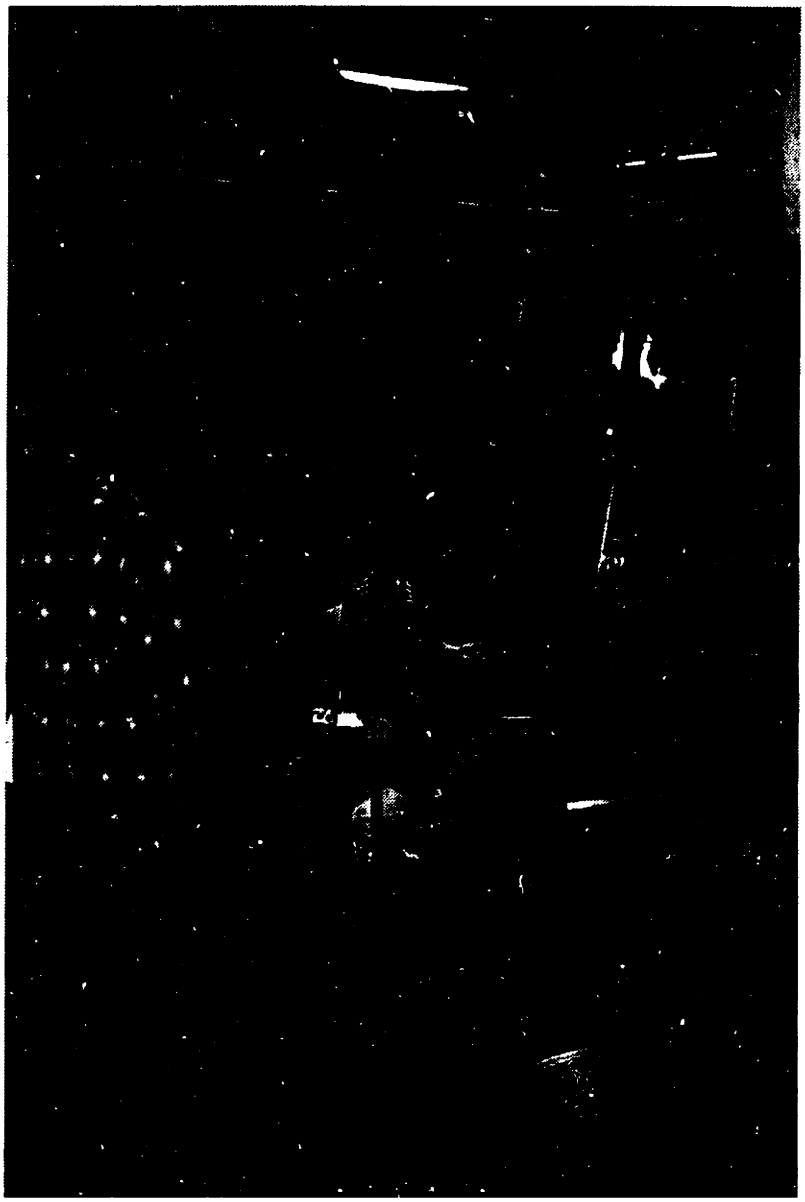


Figure 5.6: *The scene was rehearsed with all actors on stage, but then each actor was actually shot separately to simplify segmentation problems.*

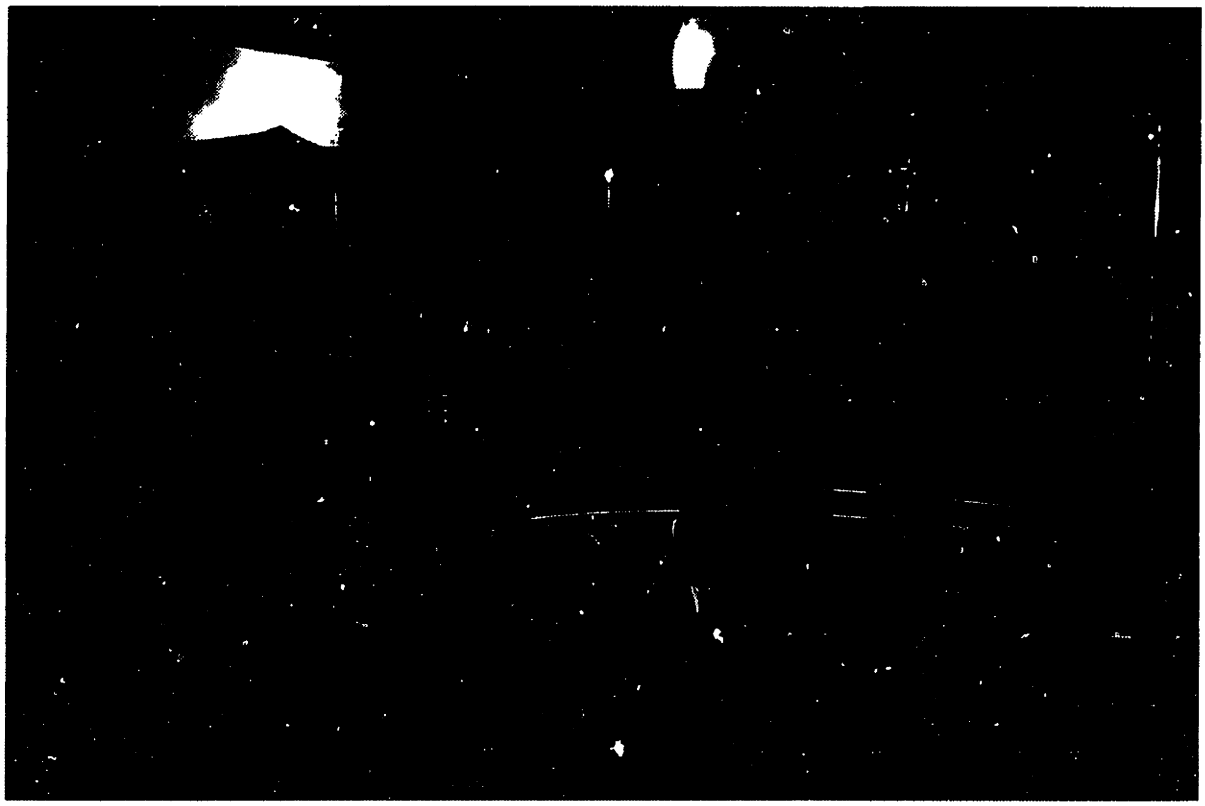


Figure 5.7: *After hours of calibrating and shooting and re-calibrating and re-shooting the master shot and several close-ups, our director (with his hands over his face) was more than a little worn out.*

blue screen but from themselves as well, which is a much more difficult problem. Simple blue screen segmentation, on the other hand, is a fairly simple process which can be easily automated. Audio was captured using both a boom and a wireless microphone and was recorded on separate tracks on one of the cameras.

After the master shot was completed, several uncalibrated close-ups were recorded with the idea that they would be useful to enhance the final presentation in key places. Shooting these close-ups with fewer cameras and without the same calibration of the master shot limits the amount of spatial movement around the virtual attic that would be possible during the times they are shown. We determined that movement during these shots was not necessary for this production and that it would be more trouble than it was worth to attempt to correlate the calibration of the master shot with that for each of the close-ups. It is also more difficult to achieve a clear three-point perspective (necessary for calibration) as the focal distance of the camera gets larger, as was needed for some of these shots.

5.4 Post-processing

Post-processing consisted of several steps. First, all the necessary video was digitized, de-interlaced, median-filtered, and temporally subsampled. Raw RGB component image sequences at 10 frames per second were obtained from this process. This frame rate was chosen to cut down on the amount of disk space needed to store everything and because we determined that the Isis structured video system on the Cheops processor would at best only be able to generate approximately 10 frames of output per second for a complicated presentation.

Next, the actors in the video were extracted from the blue background using a digital chroma-key process, producing a fourth transparency (alpha) channel in addition to the three component color channels. Once segmented, it was possible to crop each frame of the video so that extraneous blue regions were thrown away and all that was left was a smaller rectangle tightly bounded around the human figure in the image (see Figure 5.8). This step

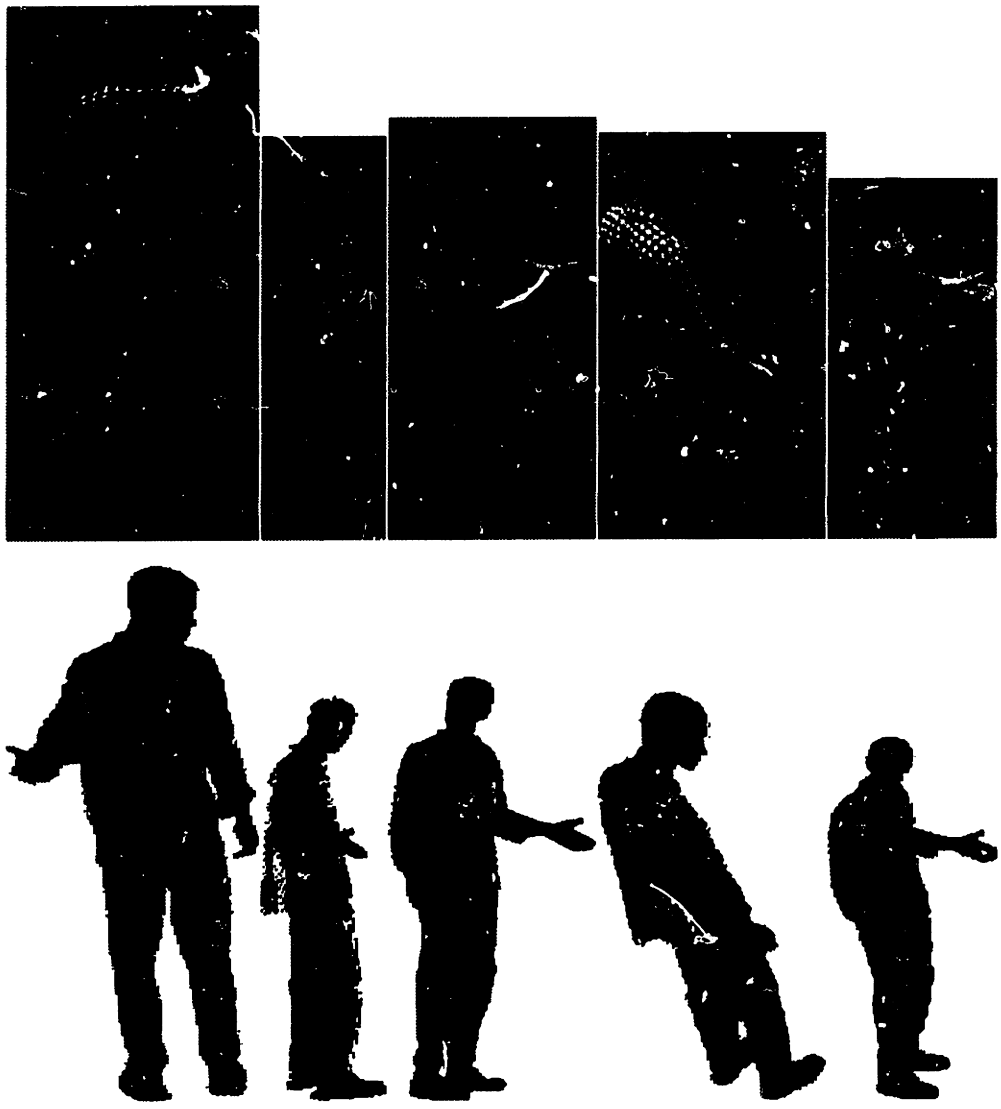


Figure 5.8: *Five views of John as captured simultaneously by the five studio cameras, and the alpha channels obtained from the chroma-key segmentation process.*

also greatly reduces the needed storage space. Special heuristics and human intervention were used to make sure that unneeded objects in the frame (such as the boom microphone) would be ignored by the part of the process that creates bounding boxes around the items of interest.

In an Isis script, when an actor is placed at a particular position in the three-dimensional space, what part of the object is placed at that point? Is it the head, the feet, the torso, or something else? Each object needs at least one positioning point, or *hotpoint*, which defines what part of the object is placed at the specified position in space. Therefore, the next post-processing step was to determine a logical hotpoint for the figure in every frame of the cropped video. The bottom of the actor's feet was an easy point to locate using some simple heuristics. Then, by projecting these hotpoints back onto the physical floor of the blue-screen stage (using the camera calibration information), it was possible to determine exactly where the actor should be placed in the virtual attic scene at every point in time. A scale factor was assigned to indicate the actual world dimensions of the object in each frame so that the actors appear to be their natural size when composited into the final scene.

The audio for the production was also digitized into a raw 44.1 KHz format and broken up into smaller "bites" that represented a single phrase or sentence of speech, thereby providing the ability to later modify the timing of each audio clip (if parts of the playout are expanded or shrunk in time).

5.5 The script

Once all the graphical and auditory media objects were in the proper format for use by the Isis delivery system, the next step was to write the actual Isis script that would deliver this interactive presentation. Several problems needed to be addressed: How could we restrict the spatial movement of the viewer? How would the presentation change to present different subjective points of view? How could we manage the master shot and close-ups in a manner that could change for different perspectives or display conditions? The approach

taken was to create a small set of high-level presentation variables that would affect playout in different ways, each of which could be represented by a single number. Some or all of these variables then would be placed under the control of the viewer.

- *Camera pose*: We decided it would be best to restrict the viewer's motion to a specific path in the attic instead of letting her roam freely and possibly get lost and miss action. Therefore, a single number can represent the viewer's "position" along this predefined path of camera parameters, as opposed to the more than 10 separate numbers that would be needed for full camera freedom. The path used was specially chosen to pass very near the camera angles that were originally captured on the blue-screen stage.

- *Closeupivity*: We needed a parameter that would express whether the playout should contain more or less close-ups—a variable that could be controlled by knowing the size of the output screen. The higher the value, the more close-ups that would be likely to be cut in at various points in the presentation. The viewer would not have the ability to move spatially during a close-up.

- *Story perspective*: Another variable was needed to express what subjective point of view of the story should be presented. Its value ranges from 0 to 9, with 0 meaning John's perspective and 9 meaning Kathy's perspective. Any value between these two extremes represents different mixtures of the two perspectives. This parameter then would affect various aspects of the narration, such as the selection and placement of close-ups and backgrounds, ambient sounds, room acoustics, etc.

- *Scene time*: This fourth variable was needed to hold the "current time" of the presentation. It could be controlled interactively or, more likely, by an internal timer.

The *timeline* construct in Isis proved invaluable for expressing the camera path and the playout of the presentation over time and over the different story perspectives. The spatial position variable indexes into a timeline (acting as a "path" line) with specific camera parameters at various points, five of which are shown in Figure 5.9. Cubic interpolation was requested to synthesize camera angles between these key points. These parameters

Figure 5.9: *The composite scene taken from five virtual camera angles near those of the five original studio cameras.*

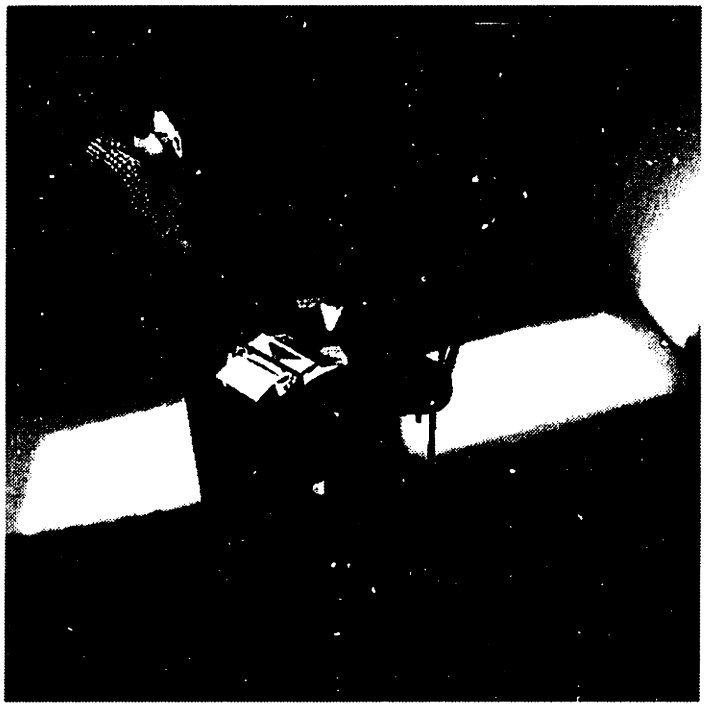


Figure 5.10: *This is a synthesized camera angle of the composite scene in that it does not correspond to any angle captured by one of the five studio cameras. This particular angle is about halfway between the real angles captured by the two high-mounted cameras.*

were used to pre-render and store several views of the three-dimensional model of the attic which are recalled and used as backgrounds before the characters are composited into the frame. Since the actors were shot from a limited number of angles, the resulting composited output might look slightly odd if the virtual camera is far from any real captured angle. As long as the camera stays within a certain "distance" of one of the real angles, this effect can usually go unnoticed (see Figure 5.10). The script determines which actor view is the most appropriate for the current camera pose.

The other three system variables are used to create a three-dimensional *story space* to determine exactly what should be showing on the screen for a given scene time, story perspective, and closeupivity setting. Nested timelines greatly simplify the creation of this space inside of the Isis interpreter. At every point in the space, indication is given of whether or not a close-up should be shown, and if it should, a frame number of a particular close-up is specified along with a background to use behind it. Other aspects of the video could be

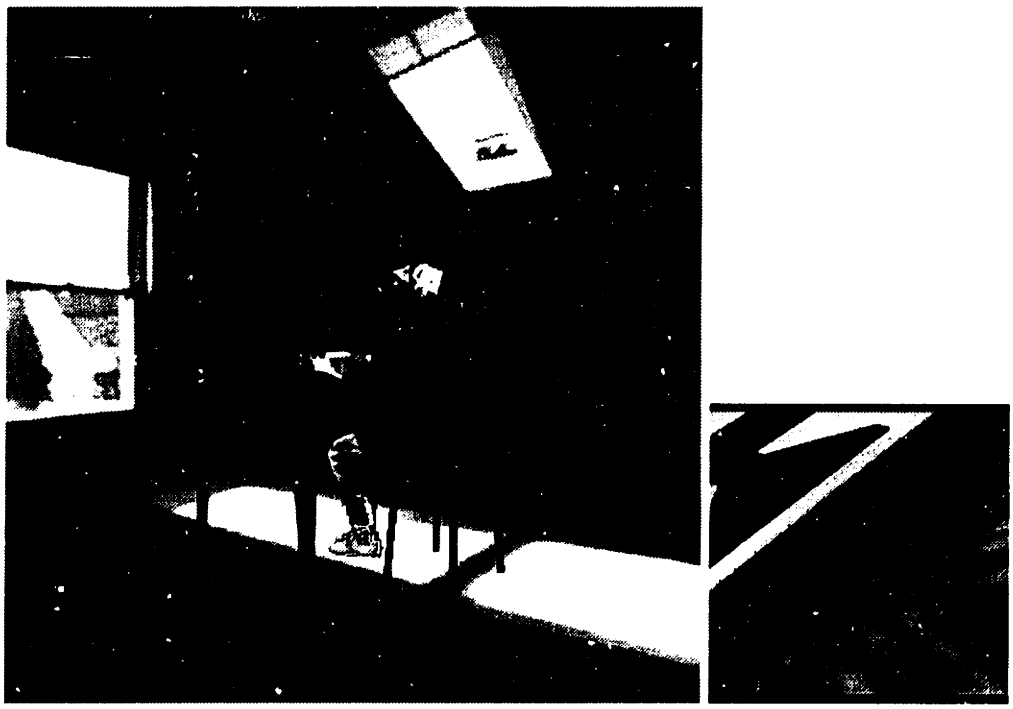


Figure 5.11: *The difference in shot selection based on display size at one moment during playback.*

controlled also, such as brightness or color tone, but the playback system does not currently support modifications to these attributes.

For example, at one particular point near the beginning of the scene, the viewer may see the master shot or a close-up of Kathy typing, depending on the size of the output window (see Figure 5.11).

More interestingly, near the middle of the scene, you might see John grabbing the rag off of the dress form in a dark gray corner of the room, or you might see the same action superimposed over a bright cloud background, or you might see John grabbing a handkerchief from the hand of a third mysterious character, all depending on the current setting of the story perspective variable (see Figure 5.12).

The virtual acoustics of the attic are smoothly interpolated between two extremes, as are the volumes of the ambient sounds in the attic. A small room with no echo or reverberation

Figure 5.12: *The difference in shot composition based on story perspective.*

is at one extreme (John's perspective), while a much larger and softer room with a lot of reverberation is at the other extreme (Kathy's perspective).

Appendix F contains an abbreviated version of the Isis script that executes the *Wallpaper* presentation, along with some comments about how it works.

5.6 The results

The final production running on the Cheops processor is surprisingly compelling, despite the fact that the entire experience is only 75 seconds long and that relatively little and highly imperfect video material is used. The presentation runs between 3 and 8 frames per second depending on the complexity of the scene, but surprisingly, many viewers have said that the "strobe" effect invoked by this frame rate actually adds to the intensity of the story.

The story perspective interaction is a complete success. The narration space created in the script gives a very cold, abusive, clausterphobic tone to the scene as perceived by John. Backgrounds are full of solid grays. The typewriter is loud and piercing, and a harsh wind sound is heard in the background. The acoustics are tuned to make everything seem very close by. Kathy's point of view is completely different however. Her world is open and free, almost as if she is not in the attic at all. Colorful cloud images are used for backgrounds in several places. The sound of the typewriter is gone, and the wind is still present, but it is more distant and is heard along with a peaceful sound of chimes. The acoustics are modified to make the room sound extremely large and contain more intense echoes and reverberation. Kathy's world is also inhabited by the character she is writing about, and consequently we see short clips of this third entity in various places.

Other aspects of the interaction, however, are less successful. The ability to move around the attic spatially is a technical achievement, but, at least in this production, there is no motivation to actually do so. One idea for improvement is to write the script such that the

subjective story perspective would also be affected somehow by the experiencer's position in the attic. At the moment though, the spatial movement interaction serves no particular purpose beyond simply demonstrating that it is possible.

The closeupivity variable turns out to have a different effect than originally planned. This variable, controlled by the size of the output window, is successful in regulating how many close-ups are shown during playout. The problem is that, in several cases, the master shot coverage is poor and it is very important to see certain close-ups, regardless of the display size, in order to enter the characters' minds and understand what is happening. When these close-ups are not shown, the experiencer is not able to fully empathize with either character, and the story consequently loses much of its power.

The user interface leaves much to be desired. We had to settle for the only input device available on Cheops—a box containing eight knobs. The story perspective, spatial position, and window size are each controlled by a separate knob. There is no visual feedback for where the knob is positioned between its maximum and minimum or for its sensitivity. A small corner graphic was later created to give some visual feedback of the current story perspective setting.

Some difficulties could have been avoided with more careful planning. The five cameras used did not reproduce colors identically (even after some were professionally tuned), resulting in visible shading differences between the various views of the actors and problems in the chroma-key segmentation process. The actress playing Kathy was unfortunately wearing tan colored pants that reflected much of the blue background, causing holes to appear in her alpha channel.

Other corners had to be cut to reduce frame generation time. We had to limit the number of on-stage actors to 3 or 4 in order to get an acceptable frame rate in some cases, and computationally intensive alpha blending was not performed, resulting in some rough edges on the boundaries between the actors and the attic background.

The audio processing system only allowed a maximum of three sounds to play simultaneously, which presented interesting challenges in trying to mix the speech of both characters with the ambient sounds (wind and chimes) and other sound effects (the typewriter). Otherwise, the three-dimensional audio system provided exactly the effects we desired for this production.

There were many places where a needed close-up was not available and had to be left out or replaced by another. It also would have been amazing to change brightness and hue dynamically during some shots or perform blending of two objects, but currently these things are beyond the capabilities of the playback system.

Since Cheops does not yet have hardware support for rendering textured objects in a three-dimensional environment, it was impossible to cast shadows of the actors as they walked around the attic. Creating realistic shadows would be possible only if the actors were all completely three-dimensional models and all the lighting characteristics of the scene were programmed into the system. Also, Cheops is only capable of scaling two-dimensional objects to certain sizes (where the numerator and denominator of the scale factor are both integers less than or equal to 7), which resulted in some noticeable aliasing of the sizes of the actors as they moved around the room.

On the whole, despite these various problems, the entire presentation is an incredible improvement, both technically and artistically, over the previous silent production, *The Museum*, which was an utterly boring experience performed by non-actors that incorporated no meaningful point of interaction.

Chapter 6

Future directions

6.1 Improving Isis

There are a great deal of opportunities to expand on the work described in this thesis at all levels. For instance, there are many improvements that could be made to the Isis language. The internal operation of the interpreter was designed carefully to be as processing-efficient as possible, but there is always room for refinement, especially in memory management. Isis is already a full programming environment, but some additional language constructs would be desirable, such as the ability to make local recursive definitions (similar to `letrec` in Scheme). One caution—many good languages and systems have often lost their edge as a result of being overly cluttered with extra and often unnecessary features and improvements. It is important to recognize that one of Isis's chief advantages is the fact that it is a *small* and *simple* language.

6.2 Enhancing structured audio and video

Other areas open to augmentation are the structured audio and video packages for Isis. The current implementation of these packages needs to be improved on the Cheops pro-

cessor. More media manipulation possibilities are needed badly, such as custom filters, transparency effects, surface texturing, shadow generation, etc. The scale factor aliasing and alpha blending problems on Cheops must be resolved as well. Other input devices besides the infamous knob box should be available.

The entire system should be implemented on architectures other than Cheops in order to expand the potential base of users and playback platforms. At the same time, more effort should be focussed on special-purpose and experimental hardware rather than ordinary general-purpose personal computers and workstations.

6.3 Increasing the potential for personalization

A recent project in the Television of Tomorrow group used structured video as a tool for personalized data visualization, specifically for weather information [Eva96]. The system collects, among other things, a list of cities and states that the user is interested in, and then it generates an Isis script that produces an customized interactive "weather report" in which the user can jump immediately to information that she needs for specific times and places. A system like this would greatly benefit from being able to store the user's preferences or common habits in order to create an optimized presentation when the system is next used. A simple standard file storage and retrieval package for Isis would provide the needed functionality.

To go further, the system could automatically read a "personality profile" at start-up and alter the media deliver in some way to best serve a particular user. For example, if the profile lists the user as having bad vision, all text and maps could be displayed in a magnified format. Or if the person has parents who live in in San Francisco, the system could always provide more detailed information about that region. A standard format for recording personal statistics and traits could be developed so that it could be retrieved and used by any application.

6.4 Expanding beyond point and click

An ongoing problem with multimedia applications is that they seem to be perpetually stuck inside of a single video screen. But Isis would also be the perfect environment for controlling productions that break free of these confines. Much more attention needs to be concentrated on experimental sensing and display devices. The mouse and the keyboard need to be thrown away completely, in favor of more interesting instruments like touch-free spatial sensors [Z95]. The entire field of interactive media is also aching for innovative output devices that go beyond simply displaying two dimensional images. Most work is still centered on visual and auditory forms, but there is very little effort, for example, on mechanisms that could interactively deliver tactile sensations.

Elegant and standard control interfaces for these kinds of devices could be built for Isis. An Isis package for creating networked media environments would also be highly desirable—large interactive installation pieces could be managed by several Isis processes running on different machines, all capable of communicating and cooperating with each other using a simple protocol.

6.5 Experimenting with storytelling

The interactive version of *The Yellow Wallpaper* only scratches the surface of what could be done in the realm of storytelling. The current state of processing and storage technology somewhat limits the ambitiousness of projects, but these obstacles are quickly being overcome. Artists must free themselves from traditional media forms and not be afraid to experiment with unusual narrative structures. There are all kinds of unexplored models of interaction that involve altering aspects of narration, such as lighting, shot composition, editing, acoustics, etc. There are also many forms that involve modifying the narrative itself and its various components. At the same time, authors must always question the appropriateness of interactivity in their works—is there a motivation to interact and is the

point of interaction meaningful, or is it just a gimmick? All in all, the opportunities for telling compelling stories in an entirely new way are endless.

6.6 Building new authoring tools

In order for experimentation to progress in a timely manner, new and better tools are needed to assist the artist in creating breakthrough pieces. Contrary to popular belief, Macromedia Director is not likely to be the solve-everything authoring tool that everyone wants—in fact there most certainly will not be a single tool that will be best for every kind of application. It is more likely that a large set of specialized authoring tools will exist, each suited to a particular kind of content and form of delivery. For example, there may be one authoring tool which is specially suited to creating applications like the *Wallpaper* production in which the point of interaction is a changing story perspective. Such tools could have the ultimate goal of outputting an Isis script that would deliver the production on a particular machine. (One such tool is described in [Tam96].) It would even be advantageous to build high-level authoring tools *inside* of Isis to support incremental updates and instant previews.

6.7 Final remarks

This thesis has addressed, at several levels, some of the problems in the authoring and delivery of interactive and personalized media. At the lowest level, an intuitive and extensible scripting environment (Isis) was designed to serve as a base for these kinds of applications. Special features of the language, such as the *timeline* storage construct and the external function call interface, greatly simplify many aspects of multimedia development. At a slightly higher level, an architecture-neutral system for creating dynamic structured audio and video productions has been implemented to operate under this scripting environment. These prototype tools emphasize expressivity and creativity, allowing authors to build complex real-time presentations by creating and controlling simple abstract components such as actors, cameras, windows, and special effects. At the highest level, important artistic

issues were highlighted through the construction of a structured video narrative using story perspective as a model for interaction.

If there was a single goal throughout this work, it was to offer some innovative ideas and solutions in the ongoing quest to bridge the gap between the artist and the latest interactive media technology. Perhaps the one sure thing that was discovered is that the potential for exciting and meaningful projects in this field is virtually unlimited.

Appendix A

Isis Reference Manual & Tutorial

A.1 Introduction

Isis, named after the ancient Egyptian goddess of fertility, is a high-level interpretive programming language which has an efficient and graceful system for interfacing with external libraries of code. Isis can be used as a stand-alone programming environment, or it can function as an interface language to packages of C functions that do specialized operations from networking and file access to graphics and sound. For example, the kinds of data structures available in Isis and its overall design make it suitable for use as an efficient scripting language for structured audio and video and other multi-media applications. This document does not describe areas where the language is applied—this is only a reference for the language itself.

Isis is another entry in a growing list of systems that serve to connect a high-level, platform-independent language with lower-level, possibly platform-dependent code libraries. Why should you use Isis as opposed to one of the other alternatives? Here are some of the advantages of Isis:

- Isis is an extensible programming language. If there is not a function available to do what you need, you can write it in the language yourself, or you can write the function in C for added efficiency and use it in Isis as you would any other function.
- The syntax of Isis is easy to learn and use. Isis has a syntax similar to that of Scheme, but *do not be fooled*—it is *not* Scheme. The language core of Isis is small compared to that of Scheme and other languages—providing a manageable set of primitive expressions and data types and allowing you to create your own higher-order entities from these basics.

- Isis is not string-based. Many popular string-based scripting languages, like TCL and Perl for example, spend much of their time converting values (especially numeric values) between their string form and a form suitable for internal processing. In Isis, this conversion is only done once at the very beginning as a script or a program is being read.

- Isis uses standard C data types for internal storage of values. Some other systems design their own data types at the bit level. In the case of `Dtypes` for example, the fundamental data types are designed to be as small as possible for network transmission. However, these special data types must be converted to and from normal C data types whenever a function from an external C library is used. This conversion phase is minimized in Isis.

- Isis is written completely in ANSI standard C, allowing it to be compiled and executed on any platform that has a standard C compiler. Isis also employs its own memory management system, thereby avoiding serious inefficiencies in dynamic memory allocation that are present on many platforms and in other systems that use the standard C or C++ memory allocation mechanisms.

- Data structures in Isis are not based on linked lists (as they are in Scheme for example). "Lists" in Isis (as you will hear about later) are formed with array-based data structures so that you have constant time access to any element in the list, greatly improving efficiency in many situations. (Actually these arrays hold *pointers* to the elements in the list, not the elements themselves, so the actual elements may be different sizes). The only problem with this method is that changing the number of elements in a list, through an "append" or a "sublist" operation for example, requires creating an array of the new size and copying the element pointers to the new array. However, this is an acceptable tradeoff since in the majority of applications, constant time reference to list items is much more important than being able to dynamically modify the lengths of large lists.

A.2 The interpreter

The interpreter for this language is a *read-evaluate-print* loop. It reads an expression from your terminal or from a file, evaluates that expression (possibly producing many side-effects), and then prints the resulting value on your terminal. If you are reading expressions from a file, then the printing of values is suppressed.

The interpreter is invoked usually with the `isis` command, although it may be different on some platforms —please read any specific information about Isis on your system for more information. Typing this command will enter you into the read-eval-print loop of the language and you may begin typing expressions to be evaluated. If you type this command

followed by a filename, then expressions will be read from that file instead of your terminal, and the interpreter will exit when the file has been completely processed.

As you type expressions, you may press return as many times as you like while in the middle of an expression. As soon as you close all parentheses that you opened, the expression will be evaluated. In addition, any text following the comment character # on a single line will be ignored.

The "Scheme mode" in the Emacs text editor is good for writing and editing script files for this language. You can configure Emacs to automatically enter Scheme mode when you open a file with a certain extension on the filename, like "`.isis`", by putting the following line in your `.emacs` file in your home directory:

```
(setq auto-mode-alist
      (append '(("\\.isis$" . scheme-mode)) auto-mode-alist))
```

A.3 Expressions, values, and types

It is important to understand the terminology of language interpreters. Something that you enter in the interpreter or that is read from a file is called an *expression*. The interpreter will *evaluate* that *expression* and return a *value*. Every *value* has a corresponding *type* associated with it, like 'integer' or 'boolean'.

The simplest kind of expression is a *constant* expression. For example if you typed the number 42 in the interpreter and pressed return, you would see this:

```
-> 42
42
->
```

In this case, you typed the *expression* `42`, which evaluated to the *value* 42 which is of *type* `Int` (an integer).

There are several *primitive* types available in the language. Each has a name, which is a way of specifying that type in the language. By convention, all type specifiers have the first letter capitalized. Here is a list of the primitive types and their names. Also listed is an example of the way a value of that type might be printed.

Value Type	Specifier	The way it is printed
integer	`Int`	`42`
real number	`Real`	`42.000000`
character	`Char`	`'q'`
boolean	`Bool`	`True`
procedure	`Proc`	`< Procedure >`
memory address	`Address`	`0x4000bb90`
structure	`Structure`	`< Structure >`
field	`Field`	`< Field >`
timeline	`Timeline`	`< Timeline >`
type	`Type`	`Real`

For brevity, the type of the value is not usually not printed along with the value itself. The *way* a value is printed gives indication of its type. Here are the rules:

- Any number printed without a decimal point is of type `Int`.
- Any number printed *with* a decimal point is of type `Real`.
- A character inside single quotes is a value of type `Char`.
- For booleans, there are only two possible values, and they are printed `True` and `False`.
- A value of type `Proc` is printed as `< Procedure >` or `< C function >`. The difference is that the former denotes that the procedure was defined in the scripting language, and the latter denotes that the procedure is a C function that is external to the interpreter. From the scripting language's point of view, these two kinds of procedures are exactly equivalent.
- A value of type `Address` is pointer to some location in the computer's memory. It will always be printed in hexadecimal form preceded by `0x`. There is no specification of what is being pointed *to*. (In C this is just a `(void *)` pointer).
- We will examine structures, fields, and timelines later, but their values are printed simply as `< Structure >`, `< Field >`, and `< Timeline >` because these types denote entities which can store a great deal of data, and to print all of it every time would be very inefficient.
- Finally, even types themselves form a type, called `Type` naturally! For example, if you enter any of the type specifiers into the script interpreter, they will be evaluated to their corresponding types. A value of type `Type` is just printed as the type name itself.

A.4 Constant expressions

Constant expressions are expressions that specify constants of the various primitive types. We have already seen how to specify an constant of type `Int`—simply type the number without a decimal point. To specify a constant of type `Real`, there must be a decimal point in the number. A `Char` constant is expressed as a character between single quotes. A `Bool` constant is expressed as either `True` or `False`. Values of type `Type` are specified by entering the type name.

We will see later how to define procedures, structures, fields, and timelines. You cannot specify an `Address` constant. Values of type `Address` will only ever be returned from and used in C functions. It could be dangerous to allow the user to specify addresses manually.

There is also a special expression `Null` which, when evaluated, produces a `<Null value>`—a value which is the absence of a value. Many operations return a null value if there is nothing else really appropriate to return.

Here are some examples of constant expressions and corresponding values:

```
-> 42
42
-> -80
-80
-> 0
0
-> 42.0
42.000000
-> -32.5
-32.500000
-> 0.0
0.000000
-> 'r'
'r'
-> 'z'
'z'
-> True
True
-> False
False
-> Real
Real
-> Char
Char
-> Type
Type
-> Null
<Null value>
->
```

There are special character constants corresponding to control characters. These special characters are entered with a backslash preceding the character. There are also pre-defined script variables set to these characters for easy reference. You can use either the literal constant or the variable to refer to these characters. The following is a list of the recognized special characters:

Character constant	Pre-set script variable
'\a'	alert
'\b'	backspace
'\f'	formfeed
'\n'	newline
'\r'	return
'\t'	tab
'\v'	vtab
'\''	squote
'\"'	dquote
'\\'	backslash

A.5 User-definable higher-order types

You can define your own types in the language by combining primitive or previously-defined types into a list. This is done via the **newtype** construct.

For example, if you enter this:

```
-> (newtype (Pos Int Int Int))
<Null value>
->
```

You have just defined a new type named **Pos**. Now to specify a value of that type, you simply enter something like this:

```
-> (Pos 10 3 42)
( Pos 10 3 42 )
->
```

This is how you specify values of type **Pos**. The way the value is printed is actually very similar to the way you typed it. Types like **Pos** can be generally called *list types* or *higher-order* types since they are collections of other types. Internally, these values are stored in array-based data structures for fast access.

Sometimes when you define a new type, you may not want to restrict its elements to be specific types. You might want to allow one or more of its elements to be *any* type. You can do this by using the **Unk** specifier in place of specific types in your call to **newtype**, like this:

```
-> (newtype (Strange Int Unk Bool))
<Null value>
-> (Strange 5 'q' False)
( Strange 5 'q' False )
-> (Strange 42 10.10 True)
( Strange 42 10.100000 True )
->
```

The **Unk** specifier used above says that the second item of a **Strange** value can be of any type. The first and third items must always be **Int** and **Bool**, respectively. Only the second item's type is unrestricted.

The **Unk** specifier can be used just like any other type in calls to **newtype**. Currently, there is no way to restrict an element to be only in a certain set of types, but this ability may come in a future version of Isis.

In forming a list, if the types of the values do not match those in the definition of the list type, you will get errors. However, the language will allow you to keep and use the incorrectly formed type. You can also turn this type-checking completely off if it annoys you, but it is a good way to make sure your values are being correctly formed. Below is what will happen if you try to make a **Pos** from 2 reals, a character, and a boolean, instead of 3 integers:

```
-> (Pos 3.2 5.0 'c' True)
* List type 'Pos', item 0 value type was not Int
* List type 'Pos', item 1 value type was not Int
* List type 'Pos', item 2 value type was not Int
* List type 'Pos', value has too many items (4)
( Pos 3.200000 5.000000 'c' True )
->
```

Errors in the interpreter will always be printed preceded by asterisks *.

You can also specify list types with an arbitrary number of items. For example:

```
-> (newtype (Poslist Pos ...))
<Null value>
-> (Poslist (Pos 3 4 5) (Pos 1 42 6) (Pos -3 0 42))
( Poslist ( Pos 3 4 5 ) ( Pos 1 42 6 ) ( Pos -3 0 42 ) )
-> (Poslist)
( Poslist )
->
```

This defines a type **Poslist**. The "..." means there will be an arbitrary number of values

of the *last* type in the list. In this case, the last (and only) type in the list is `Pos`, so we have defined a type which is a list of an arbitrary number of `Pos` values (including possibly zero values).

You can "repeat" the last N items in the list like this by putting a number right after the "..." with no space, like this:

```
-> (newtype (Weirdo Char Bool Int Real ...3))
<Null value>
-> (Weirdo 'q' True 4 5.0 False 10 -3.4 True -1 -42.0)
( Weirdo 'q' True 4 5.000000 False 10 -3.400000 True -1 -42.000000 )
->
```

This defines type `Weirdo`—a `Char` followed by as many sequences of `Bool Int Real` as you like (even zero).

You can also define a type to be equivalent to a previously defined type by entering:

```
-> (newtype Vec Pos)
<Null value>
-> (Vec 1 2 3)
( Vec 1 2 3 )
->
```

This defines the type `Vec` to be the same as type `Pos`.

Some frequently used list types are built in. Here is a list of them along with how they would be defined manually if they were not built-in:

List type name	How it is defined
`List`	`(newtype (List Unk ...))`
`IntList`	`(newtype (IntList Int ...))`
`RealList`	`(newtype (RealList Real ...))`
`BoolList`	`(newtype (BoolList Bool ...))`
`AddrList`	`(newtype (AddrList Address ...))`
`String`	`(newtype (String Char ...))`

Again, these types are already defined, so you do not need to define them yourself. Type `List` is simply an arbitrary length list of values of *any* type. The others are lists of values of specific types. Here are a couple examples:

```
-> (List 45.3 'e' (Pos 1 2 3) True)
( List 45.300000 'e' ( Pos 1 2 3 ) True )
-> (IntList 34 45 56)
( IntList 34 45 56 )
-> (BoolList True False False True)
( BoolList True False False True )
-> (BoolList)
( BoolList )
->
```

To specify **String** values, you could type something like (**String** 'h' 'e' 'l' 'l' 'o'), but typing out each individual character could get tedious. Therefore there is a SHORT CUT to specifying **String** values—just type the string between double quotes. The interpreter will automatically put the characters into a list of type **String**. Also, the interpreter will print out any value of type **String** as a string between double quotes, just like you would type it in. This provides a convenient way to specify strings but to still have them stored in the list based structures so you can take advantage of all the list modification routines to be discussed later.

```
-> (String 'h' 'e' 'l' 'l' 'o')
"hello"
-> "hello"
"hello"
```

A.6 Variables

You can set your own variables to any value using **set**. Here are some examples:

```
-> (set a 42)
42
-> (set goodpos (Pos 1 2 3))
( Pos 1 2 3 )
-> a
42
-> goodpos
( Pos 1 2 3 )
-> (set dumdum (Pos a 34 a))
( Pos 42 34 42 )
->
```

A.7 Other expressions

A.7.1 `if` expression

(`if` *cond-exp then-exp* [*else-exp*])

This works like the classic `if` statement—if the *cond-exp* evaluates to a "true" value, then the result of evaluating the *then-exp* is returned. Otherwise the result of the *else-exp* is returned. Whether or not a value is true depends on its type. If it is a `Bool`, then it is only true if its value is `True`. If it is a number, address, or character, any non-zero value is considered true. A `<Null value>` is considered false, as is any other null pointer that might be encountered.

You may omit the *else* part of the statement if you wish—Isis will assume it to be `Null`. Only one of either the *then-exp* or the *else-exp* (if it exists) is ever evaluated.

A.7.2 `cond` expression

(`cond` (*cond-exp value-exp*) (*cond-exp value-exp*) ...)

The *cond-exp*s are evaluated in sequence until one of them evaluates to a "true" value, and then the corresponding *value-exp* is evaluated and its value is returned. Only the *value-exp* corresponding to the first true condition is evaluated—the rest are left untouched. Also, if there are remaining *cond-exp*s after finding the first true one, they are left untouched as well. A default condition of `True` can be used as the last *cond-exp* to specify a default action.

A.7.3 `while` expression

(`while` *cond-exp body-exp*)

In this case, the *body-exp* is continually evaluated as long as the *cond-exp* evaluates to a true value. The value of the entire `while` expression is the value of the *cond-exp* that caused the loop to stop.

A.7.4 `switch` expression

(`switch` *switch-exp* (*case-exp value-exp*) (*case-exp value-exp*) ...)

The *switch-exp* is evaluated, and its value is compared in sequence to the values of the *case-exps*. If the two values match, the corresponding value of the *value-exp* is returned. If they do not match, the *value-exp* is not evaluated. If no match is found, a `<Null Value>` is the result.

A.7.5 `begin` expression

(`begin` *exp exp exp* ...)

Each expression is evaluated in sequence. The value of the final expression is returned.

A.7.6 `let` expression

(`let` [(*variable-name exp*) (*variable-name exp*) ...] *body-exp*)

The `let` expression sets up a *local environment* in which each *variable-name* is bound to the value of its corresponding *exp*. The variable names may be the same as other variables in the parent environment, but when you refer to a duplicated variable name inside the `let` expression, you will be referring to the local variable, not the one in the parent environment. All other variable names will refer to variables in the parent environment. The *body-exp* is evaluated within the *scope* of this new environment, and its value is returned.

Examples of these expressions in use are found later.

A.8 Procedure application and built-in procedures

Syntax:

(*exp exp exp* ...)

This is a simple model of procedure application. The first expression is evaluated. If its

value is a procedure, then the rest of the expressions are evaluated, and then the procedure is applied to those values. For example, to apply the built-in procedure +, try the following:

```
-> (+ 3 4 42)
49
->
```

There are tons of built-in procedures, each of which is described below:

A.8.1 +, -, *, /, %

These procedures each take a variable number of arguments and apply the corresponding mathematical operations in sequence to the values. % is the modulo operator. You can perform these operations on a mixture of `Real` and `Int` values, or even lists containing `Real` and `Int` items! Note: in each case, the result of the application will be the same type as the *first argument* given.

```
-> (+ (Pos 3 4 5) (Pos 54 -25 11))
( Pos 57 -21 16 )
-> (+ 3.4 2 6.5)
11.900000
-> (+ 3 4.5 6.7)
13
->
```

A.8.2 abs, sin, cos, tan, asin, acos, atan2, sinh, cosh, tanh, exp, log, log10, pow, sqrt, ceil, floor, deg->rad, rad->deg

These procedures all perform the same operations as the standard C functions by the same name. `deg->rad` and `rad->deg` are the only non-standard ones—they perform degree to radian conversion. The arguments can be `Int` or `Real`.

```
-> (cos (deg->rad 30))
0.866025
-> (sin (deg->rad 30))
0.500000
-> (rad->deg (atan2 (/ (sqrt 2) 2) (/ (sqrt 2) 2)))
45.000000
->
```

A.8.3 =, !=, <>

These functions check equality or inequality of 2 values of *any* type. Result is either `True` or `False`. `<>` is the same as `!=`.

A.8.4 <, <=, >, >=

These inequality operations only work on numbers (`Int` or `Real`), not lists. Result is either `True` or `False` again.

A.8.5 and, nand, or, nor, not

These perform the specified logical operations on a variable number of arguments.

A.8.6 ref, head, tail

These return a specific item in a list. `(ref k list)` returns the k'th item in the list. `(head list)` returns the first item in the list. `(tail list)` returns the last item in the list.

SHORT CUT: As a short cut to `(ref k list)`, you can use the syntax `(list k)`. This looks like a procedure application, but since the first item is a list and not a procedure, the interpreter knows to perform the list reference operation. Here's an example:

```
-> (set nicepos (Pos 3 5 7))
( Pos 3 5 7 )
-> (ref 1 nicepos)
5
-> (nicepos 1)
5
->
```

A.8.7 length

`(length list)` returns the number of items in the list.

A.8.8 `insert-before, insert-after, head-insert, tail-insert`

These functions insert items into a list. `(insert-before k val list)` inserts the value before the `k`'th item in the list. `(insert-after k val list)` inserts the value after the `k`'th item in the list. `(head-insert val list)` inserts the value at the beginning of the list. `(tail-insert val list)` inserts the value at the end of the list.

A.8.9 append

This function appends 2 or more lists together. The resulting list is the type of the first argument.

A.8.10 `sublist, first, last, allbutfirst, allbutlast`

These functions operate on a list value and all return a specific part of that list argument. `(sublist n m list)` returns the part of list between the `m`'th and the `n`'th item, inclusive. `(first k list)` returns the first `k` items in the list. `(last k list)` returns the last `k` items in the list. `(allbutfirst k list)` returns all but the first `k` items in the list. `(allbutlast k list)` returns all but the last `k` items in the list. Indexing starts at 0 in all list operators.

Examples of some list operations:

```
-> (set niceplaces (Poslist (Pos 3 4 6) (Pos 1 -2 -3)))
( Poslist ( Pos 3 4 6 ) ( Pos 1 -2 -3 ) )
-> (length niceplaces)
2
-> (niceplaces 0)
( Pos 3 4 6 )
-> (length (niceplaces 0))
3
-> (set niceplaces (tail-insert (Pos -42 -43 -44) niceplaces))
( Poslist ( Pos 3 4 6 ) ( Pos 1 -2 -3 ) ( Pos -42 -43 -44 ) )
-> (sublist 1 2 niceplaces)
( Poslist ( Pos 1 -2 -3 ) ( Pos -42 -43 -44 ) )
-> (head (allbutfirst 2 niceplaces))
( Pos -42 -43 -44 )
-> (tail (head (allbutfirst 2 niceplaces)))
-44
->

-> (length "hello")
5
-> (set worldstr "world")
"world"
-> (set blahstr (append "hello " worldstr))
"hello world"
-> (sublist 3 7 blahstr)
"lo wo"
-> (append (first 4 blahstr) (allbutfirst 5 blahstr))
"hell world"
->
```

A.8.11 seed-random, random

These functions control the random number generator. The pre-defined script variable **rand-max** is the highest random number that will ever be generated. Call **seed-random** with any number you like. Then call **random** with no arguments to get a random integer between 0 and **rand-max**. Call **random** with one number to get a random integer between 0 and that number (inclusive). Call it with 2 numbers to get an integer between those two numbers (inclusive).

```
-> rand-max
32767
-> (seed-random 42)
42
-> (random)
19081
-> (random 10)
5
-> (random 42 69)
55
->
```

A.8.12 print, display

These functions print things. Each takes as many arguments as you like. The arguments are printed in the sequence entered. The **print** function prints only **String** and **Char** values, without their corresponding double and single quotes. The **display** function displays any value in the same fashion that the interpreter would print it—**Strings** and **Chars** will have surrounding quotes. Newlines must be explicitly entered in **print**, but a newline is automatically placed at the end of a call to **display**.

```
-> (print "The museum movie" newline "in Structovision" newline)
The museum movie
in Structovision
<Null value>
-> (begin (print "The value of the day is ") (display (Pos 3 2 4)))
The value of the day is ( Pos 3 2 4 )
<Null value>
-> (print "This is a string printed with 'print'.\n")
This is a string printed with 'print'.
<Null value>
-> (display "This is a string printed with 'display'.\n")
"This is a string printed with 'display'.
"
<Null value>
->
```

A.8.13 load, run, interactive, stdin, done, exit, quit, end

(done), **(exit)**, **(quit)**, and **(end)** all have the same effect of indicating the end of the script file or the end of the interactive session.

`(load filename)` or `(run filename)` reads the script specified by the filename which must be of type `String`. Items in the file are read and evaluated just as if you had typed them yourself. When the end of file is reached, control is returned back to where the call was made.

`(interactive)` causes the interpreter to read info from the standard input. It can be used in a script that is being read from a file to cause the input to switch temporarily to the terminal. When the interactive session is ended by a `(done)`, `(exit)`, `(quit)`, and `(end)`, control is returned back to the script being read and processing there continues.

`(stdin)` switches the input file of the interpreter from whatever it was to the standard input, thereby ignoring the rest of a file that was being read at the time.

A.8.14 istype

`(istype <type> <item>)` returns `True` if the item evaluates to a value of the specified type, and `False` otherwise. It only performs a type query—it does not check if the value is correctly formed (done by the `typecheck` function described later).

A.8.15 env, types

`(env)` prints information about every binding in the top level environment of the interpreter. `(types)` prints information about all of the types that have been defined in the interpreter up to that point.

A.8.16 print-values, set-typecheck, typecheck

`(print-values False)` turns off the printing of values in the read-eval-print loop. Passing `True` turns it on again. Similarly, `(set-typecheck False)` turns off list value type checking and passing `True` will turn it on again.

`(typecheck <item>)` typechecks the item's value against the original type definitions and prints any errors in the formation of the value.

A.8.17 cast

This rarely-used function can be used to cast values of one type to another type. It takes 2 arguments—first the type to cast to, and then the value to be cast. The cast is performed if it is possible. `Int` can be cast to `Real` and `Real` can be cast to `Int`. List values can be cast to any equivalent list type. `Real` and `Int` can be cast to and from `Bool`. Any other casts may result in an error.

```
-> (cast Int 4.5)
4
-> (cast Real 42)
42.000000
-> (cast Vec (Pos 3 5 7))
( Vec 3 5 7 )
-> (cast Bool 123)
True
-> (cast Bool 0.0)
False
-> (cast Int False)
0
-> (cast Real True)
1.000000
->
```

A.9 Procedure definition

You can define your own procedures using the following syntax:

(`proc` (*formals*) *body-exp*)

The *formals* are the arguments for the procedure. The *body-exp* is the expression that will be evaluated within the *scope* of the environment the procedure is defined in, plus the formal parameter bindings. The result of the evaluation is returned as the result of the procedure application.

```
-> (set add2 (proc (x) (+ x 2)))
< Procedure >
-> add2
< Procedure >
-> (add2 42)
44
->
```

You can create recursive procedures if they are defined at the top level environment (not within a `let` expression). For example, here is a recursive factorial function:

```
-> (set fact (proc (a) (if (<= a 1) 1 (* a (fact (- a 1))))))
< Procedure >
-> (fact 10)
3628800
->
```

A.10 C functions

One of the most powerful features of this language is that in addition to defining procedures inside the script interpreter, you can also define your own C functions and use them in the interpreter just as you would any other procedure. From the scripting language's point of view, applications of those C functions work in exactly the same way as applications of script procedures. When you apply procedures in your scripts, you might be calling C functions or script procedures—and you would never know the difference. In fact, all of the primitives described above are implemented as C functions.

C functions can be written if a highly efficient routine is needed for some part of the program. They can also be used to interface this language with externally developed C packages, like graphics and sound libraries or user interface software. The structured video and sound package for this language is being developed using the C function interface.

Defining C functions for use in the interpreter is a fairly simple process. The arguments to your C function are passed in a standard argc/argv format where each argument is a "value" from the script interpreter. You will need to translate the interpreter's value format to your own integers or characters or whatever else you use. Special C macros are provided to make this very easy. You then prepare a "value" to be returned to the interpreter. Finally, you register the function with the script interpreter so that it will become a part of the default top-level environment when the interpreter is started up. Then you can use it like any other procedure!

Another document describes this process in more detail.

A.11 Structures

So far you have seen how you can store data in lists. Lists are an efficient way of storing ordered unnamed pieces of data. However there may be situations where you might like to have something like a C structure for storing data. This language provides that through the `Structure` data type. A `Structure` is an unordered collection of named fields, each of which contains a value.

You create a structure with `new-structure`, like this:

```
-> (set soundstrc (new-structure))
< Structure >
->
```

This newly-created structure is completely empty. It has no fields yet. You create fields inside the structure by using the `add-field` procedure. In order to find the field later, you must give each new field a "reference value" which can be any Isis value you like. This value acts as a name for the field.

```
-> (add-field soundstrc 42)
< Field >
-> (add-field soundstrc 'f')
< Field >
-> (add-field soundstrc "position")
< Field >
->
```

Now this structure contains 3 fields with reference values `42`, `'f'`, and `"position"`. You obtain access to the fields by using the return value of `add-field`, or with the `get-field` function, which attempts to find the field corresponding to a particular reference value inside a structure. If a field with that reference exists, it is returned in a value of type `Field`. You can then use this field in the `set-field` function to set the value of the field, or you can use it in the `fieldval` function which gets the value of the field.

```
-> (set sound-filename-field (get-field soundstrc 'f'))
< Field >
-> (set-field sound-filename-field "/mas/garden/eeny/soundfile")
"/mas/garden/eeny/soundfile"
-> (fieldval sound-filename-field)
"/mas/garden/eeny/soundfile"
->
```

There are two convenient SHORT CUTS to mention here. Instead of typing

`(get-field structure refval)`, you can type simply `(structure refval)`. This looks like a procedure application, but the first item in the application is a structure instead of a procedure, so the interpreter treats this as a call to `get-field`. Similarly, instead of typing `(fieldval field)`, you can type just `(field)`, and this will return the value of the field just as if you were calling `fieldval`.

```
-> (set-field (soundstrc "position") (Pos 3 2 1))
( Pos 3 2 1 )
-> ((soundstrc "position"))
( Pos 3 2 1 )
->
```

The reason structures and fields are conceptually separate entities is to allow you to reference a particular field directly after you've found it just once, rather than repeatedly calling `get-field` (an operation which could require many value comparisons to find the desired field).

A.12 Timelines

Finally, we have arrived at one of the most useful innovations of this language. In addition to simple lists types and structures, there is a third kind of data structure called a `Timeline`. When you first create a timeline, it is completely empty. You then place values at any real-numbered point, or time, on the timeline. When you enter each point's value, you also specify whether you want the value of the timeline to be interpolated between that point and the point directly before it on the timeline. You can specify either linear or cubic interpolation. The timeline therefore has a "value" at every real-numbered point in time!

For the purposes of the following examples, let's define a new type called `RPos` which will be a list of 3 real numbers instead of 3 integers.

```
-> (newtype (RPos Real Real Real))
<Null value>
->
```

To create a timeline, use the `new-timeline` function. This function will accept one argument which will be used as the default value of the timeline in case there is no information for a particular part of the timeline.

```
-> (set dave-position (new-timeline (RPos 0.0 0.0 0.0)))
< Timeline >
->
```

To insert values at *key* points on the timeline, use the **key** function. It takes 3 or 4 arguments. The first is the timeline itself, the second is the real-numbered time you want to insert at, the third is the value you wish to insert there, and the fourth is an optional interpolation flag.

The times specified in timeline operators can be **Int**s or **Real**s (they are converted automatically to **Real**s internally).

There are two built-in script variables: **linear** and **cubic**, which can serve as the interpolation flag. The interpolation flag says how you would like the value of the timeline to be interpolated between this point on the timeline and the key point directly before it on the timeline. The earlier key point may be defined at any time—the order in which you insert points in the timeline has no effect on its interpretation.

If you do not want interpolation, leave out the optional argument. In that case, the value of the timeline between the time being inserted at and the earlier key point will be just be held at the value of the earlier key point.

You find the value of the timeline at a certain time by calling the **keyval** function. It takes the timeline as the first argument, and the time for which you want the value as the second argument.

```
-> (key dave-position 0 (RPos 2.0 4.0 6.0))
( RPos 2.000000 4.000000 6.000000 )
-> (key dave-position 20 (RPos 10.0 42.0 85.0) linear)
( RPos 10.000000 42.000000 85.000000 )
-> (key dave-position 25 (RPos 0.0 68.0 70.0) linear)
( RPos 0.000000 68.000000 70.000000 )
-> (keyval dave-position 10)
( RPos 6.000000 23.000000 45.500000 )
-> (keyval dave-position 4.6793)
( RPos 3.871720 12.890670 24.483235 )
-> (keyval dave-position -42)
( RPos 0.000000 0.0000000 0.0000000 )
->
```

Notice that asking for the value at time -42 results in the default value of the timeline.

You can delete key points from the timeline using the **delkey** function which will delete the closest key point less than or equal to the given time. It also returns the value that was deleted:

```
-> (delkey dave-position 27)
( RPos 0.000000 68.000000 70.000000 )
->
```

When cubic interpolation is used, there must be at least 2 key points before and after the time you want the value for in order for cubic interpolation to be possible. There is no wraparound in determining the points to use in the interpolation.

There is another convenient SHORT CUT here. Instead of entering
`(keyval timeline time)`, you can just enter `(timeline time)`. The interpreter recognizes that the first item in this application is a timeline, and therefore treats it as a call to `keyval`.

```
-> (key dave-position 30.0 (RPos -34.0 -40.1 -234.4) cubic)
( RPos -34.000000 -40.100000 -234.400000 )
-> (key dave-position 40.0 (RPos 23.3 354.3 123.4) cubic)
( RPos 23.300000 354.300000 123.400000 )
-> (dave-position 20.3)
( RPos 9.369685 41.024039 80.745221 )
-> (dave-position 28.5)
( RPos -32.391944 -52.456888 -219.376075 )
->
```

The values in the timeline do not have to be numeric, nor do they have to be of the same type. However, interpolation will only occur if the key points involved are of the *same type*, and are numeric in some way. If a list type combines numeric and non-numeric types, only the numeric parts will be interpolated. No example of this is given, so please try it yourself to see if it works!

Two special functions are provided to help deal with timelines. They are `key-prev` and `key-next`. Calling `(key-prev timeline time)` will return the highest time of a key point on the timeline which is less than the specified time. `(key-next timeline time)` will return the lowest time of a key point on the timeline which is greater than the specified time. If you call `key-prev` with a time less than or equal to the lowest defined key time in the timeline, the return value will be the highest defined key time in the timeline. Similarly, if you call `key-next` at the highest point in the timeline, you will be returned the lowest point. This gives you a way to traverse all the defined key points on a timeline should you need to.

```
-> (key-prev dave-position 34)
30.000000
-> (key-next dave-position 34)
40.000000
-> (key-next dave-position 100)
0.000000
-> (key-prev dave-position 20)
0.000000
-> (key-prev dave-position 0)
40.000000
->
```

Also, explicit linear and cubic interpolation functions are provided so you can do interpolation outside of timelines if you like. The functions are `linear-interpolate` and `cubic-interpolate`. The first argument of each is a `Real` between 0.0 and 1.0 specifying the location between the two values for which the value is desired. `linear-interpolate` takes 2 more arguments (the values to be interpolated), and `cubic-interpolate` takes 4 more (interpolation occurs between the 2 middle values). The type of the result will be the same as the type of the value arguments.

```
-> (linear-interpolate .4 0.0 10.0)
4.000000
-> (cubic-interpolate .4 4.0 0.0 10.0 7.0)
3.616000
-> (cubic-interpolate .4 100.0 0.0 10.0 -300.0)
11.440000
->
```

Pretty cool eh?

A.13 More examples

A.13.1 Countdown

```
-> (let ((count 10))
      (begin (while (> count 0)
                (begin
                  (display count)
                  (set count (- count 1))))
      (print "Detonation!" newline)))
10
9
8
7
6
5
4
3
2
1
Detonation!
<Null value>
->
```

A.13.2 Fibonacci sequence

```
-> (set fibseq
     (proc (k)
       (let ((ct 0) (fib1 1) (fib2 0) (fibnow 2))
         (begin
           (display 1)
           (while (< ct k)
             (begin (set fibnow (+ fib1 fib2))
                    (display fibnow)
                    (set fib2 fib1)
                    (set fib1 fibnow)
                    (set ct (+ ct 1))))
           Null))))
< Procedure >
-> (fibseq 20)
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
<Null value>
->
```

A.13.3 Y combinator

```
-> (set y (proc (m) ((proc (future)
                           (m (proc (arg)
                                    ((future future) arg))))
                     (proc (future)
                           (m (proc (arg)
                                    ((future future) arg)))))))
< Procedure >
-> (set m
     (proc (recfun)
           (proc (x)
                 (if x
                     (* x (recfun (- x 1)))
                     1))))
< Procedure >
-> (set fact (y m))
< Procedure >
-> (fact 5)
120
->
```

Appendix B

Structured Video with Isis

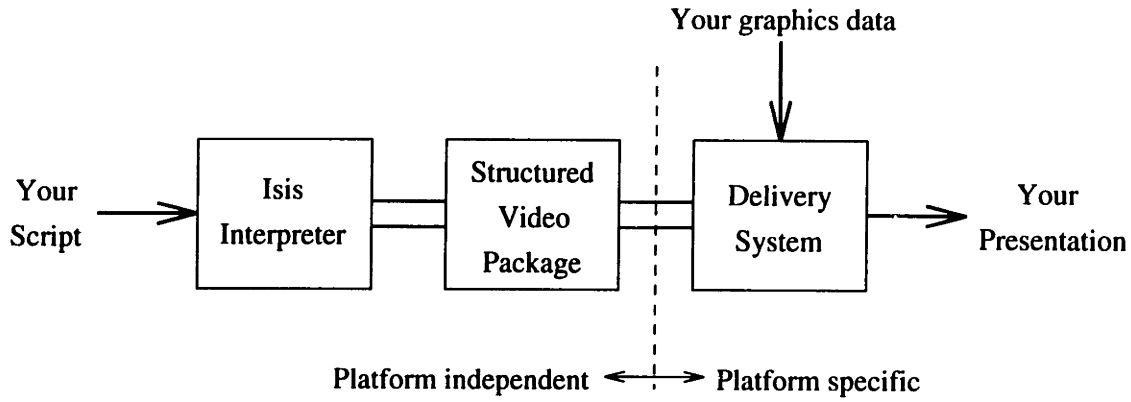
B.1 Introduction

This document will explain how to create structured video presentations using the *Isis* scripting language. You should first read the document *Isis: A multi-purpose interpretive scripting language* which explains the details of the Isis language itself.

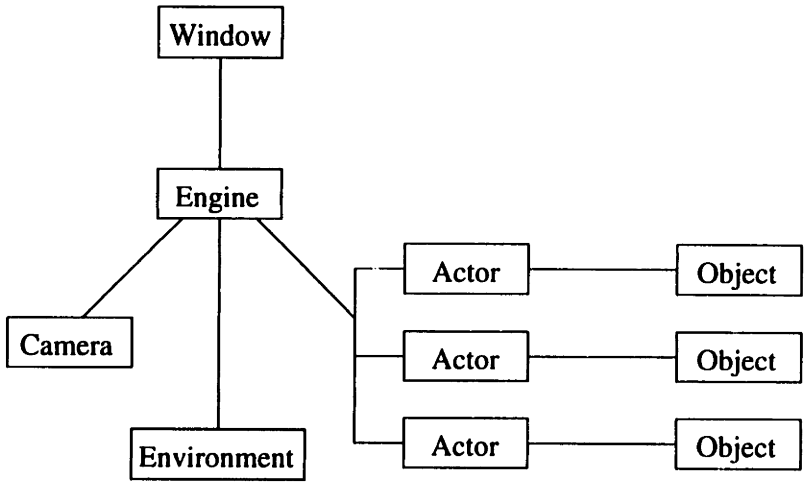
What is *structured* video? Traditional video is represented in terms of pixels, scanlines, and frames. On the other hand, *structured* video, or *object-oriented* video, is represented in terms of several component *objects* that are combined in some manner to produce the video scene. These objects may be of many different kinds—two dimensional images or movies, three-dimensional models, text, graphics, etc. They are manipulated and assembled together in different ways over time to generate each frame of a presentation. Representing video in this way tends to greatly simplify the problems of interactive and personalized delivery.

The package of Isis functions described in this document will allow you to express highly interactive structured video presentations in a platform-independent manner. Since Isis is an interpretive scripting language, you can view and change aspects of your presentation instantly without having to recompile entire programs each time you make a small modification. You will be able to use delivery systems on several different platforms to view and interact with your production. Each platform will offer advantages over the others. For example, Isis and the structured video package running on the Cheops processor will provide real-time playback and interaction with fair image quality, whereas the system running on an Alpha or other general purpose workstation will offer enhanced output quality but will run at a much reduced frame rate with only intermittent interactivity.

Let's begin with a general overview of the design of the system. There are 3 main segments. At the highest level, there is the platform-independent Isis script interpreter. The structured video package is then accessed in Isis merely by calling various built-in Isis functions. These functions are also machine-independent, so it is possible for you to write a single script that will run on any platform. It is then the job of the structured video package to use machine specific routines and libraries to deliver your presentation as effectively as possible on your platform.



The structured video package lets you create and manipulate six different kinds of abstract entities: objects, engines, actors, cameras, environments, and windows.



Essentially, consider yourself as the *camera*, the other people and things in the production as the *actors*, and the details of how you perceive those people and things as the *environment*. The *window* describes how what the camera sees will be translated to a window on your screen. The *engine* can loosely be thought of as the *translator* from the virtual world of the

presentation to the output device in the real world. Using the information stored in a certain camera, an environment, one or more actors, and a window, the engine does all the work to build frames of a presentation. You can have more than one engine in a presentation, each of which can share actors, cameras, etc., with any of the other engines. This gives you the power to control several presentations in many different windows simultaneously.

Each actor must point to an *object*, which is an entity that refers to actual graphics data, such as an image, a movie, a three-dimensional model, a font, etc. More than one actor can use the same object simultaneously, but each actor can only point to a single object at a time. Only raw graphics data (or information about where it is located) is stored in the *object*. Information about how that data gets rendered is part of the *actor*.

The functions available to you in Isis allow you to create instances of these six kinds of entities and modify parameters inside each of them. A typical script will begin by calling initialization functions and creating all the needed entities. To output a each frame of a presentation, the script need only set the appropriate parameters inside each entity to the desired values (possibly taking into account user input) and then call a function to actually draw the frame. If this "update and draw" routine is performed repeatedly, it is possible to achieve essentially real-time output! The following sections of this manual will explain this process in greater detail.

B.2 Running Isis and the Structured Video Package

In order to use the functions in the structured video package, you first have to check that the version of Isis you are using is linked to the package. When you start Isis, you will see something like this:

```
*** Isis (version 2.0)
*** by Stefan Agamanolis
+++ Structured Video Package
=== Structured Audio Package
--- User Interface Package

->
```

If you don't see "`Structured Video Package`" listed here, then you must use a different version of Isis. Ask your friendly Isis software maintainer where to find the correct version.

Next, you must load the file `structured-av.isis` which should be located in the same place as the Isis interpreter program, or in a directory where standard Isis scripts are stored. Although not required, this file contains helpful type definitions that will simplify

using the structured video system. Make sure to specify the entire pathname if necessary. Otherwise, if the file `structured-av.isis` is in your current directory, you can simply type:

```
-> (load "structured-av.isis")
Reading from script `structured-av.isis'...
13 items processed
<Null value>
->
```

For reference, this file makes the following type definitions:

```
# structured-av.isis
# Helpful type definitions structured audio and video scripts

(newtype (Pos Real Real Real))
(newtype (Vec Real Real Real))
(newtype (Dim Real Real Real))

(newtype (Loc Int Int))
(newtype (Size Int Int))

(newtype (Vec4 Real Real Real Real))
(newtype (Matrix4 Vec4 Vec4 Vec4 Vec4))  # each vector is a ROW

(newtype (Rect Int Int Int Int))
(newtype (RealRect Real Real Real Real))

(newtype (View Vec Vec Vec Real RealRect))

(newtype (Poslist Pos ...))
(newtype (Viewlist View ...))
```

B.3 Initializing and controlling system state

```
(initialize-video-system)
(start-video-system)
(stop-video-system)
(reset-video-system)
(terminate-video-system)
```

The first package function you must call in any script that uses structured video is `initialize-video-system`. This function should only ever be called once. Then, before

outputting any frames from any engine, you must call `start-video-system` which will open any windows that you have created and prepare them for output. `stop-video-system` will close any open windows but does not destroy them—they can be reopened with another call to `start-video-system`. `reset-video-system` allows you to re-initialize the system to the same state as when the system was first initialized. `terminate-video-system` will stop all video output, close all windows, and shut down any external processes associated with the delivery system. It should only ever be called once, usually at the very end of a script.

```
-> (initialize-video-system)
<Null value>
-> (start-video-system)
<Null value>
->
```

B.4 Creating structured video entities

```
(new-video-engine)
(new-window)
(new-camera)
(new-video-environment)
(new-actor)
(new-video-object data-filename internal-name class-name)
```

These functions will create new structured video entities. Each returns a value of type `Address` which you will use to refer to the entity later, so you should always be sure to store it (usually by assigning it to a variable).

```
-> (set myengine (new-video-engine))
0x402d4000
-> (set win1 (new-window))
0x402d4020
-> (set env1 (new-video-environment))
0x402d4040
-> (set cam1 (new-camera))
0x402d4060
-> (set act1 (new-actor))
0x402d4080
->
```

Three `String` arguments are required by `new-video-object`:

- `data-filename` should be the filename or URL where the graphics data may be found. URL support may or may not be present on your system. The format of the data is inferred from its filename or extension.
- `internal-name` can be whatever you like as it is only used in printing debugging information about the object, not in any processing.
- `class-name` is used to group certain objects in order to aid the caching mechanism in the delivery system. For example, if there are several objects which are used as backgrounds but only one is ever used at any one time, then you should give these objects the same class name, perhaps something like `"background"`. Do the same for other groups of objects, giving each group a unique class name.

```
-> (set room1obj
     (new-video-object "/grad/stefan/room1.rgb" "Room 1" "background"))
0x402d40a0
-> (set room2obj
     (new-video-object "/grad/stefan/room2.rgb" "Room 2" "background"))
0x402d40c0
-> (set personobj
     (new-video-object "/grad/stefan/person.rgba" "Person" "foreground"))
0x402d40e0
->
```

B.5 Updating parameters

```
(vid-update entity parameter value parameter value ...)
```

There are several parameters inside each structured video entity, each of which has a value that you may change to alter the state of that entity in some way. The parameters are given default values when the entity is first created. It is your job to set these parameters appropriately before calling the function that plots a frame. The way you set these parameters is by calling the `vid-update` function. This function accepts a variable number of arguments. The first argument must be the entity you wish to set parameters within. Then you pass a parameter specifier followed by the value you wish to give that parameter. You may pass as many parameters and corresponding values in a single call as you like. The examples in the following sections should make this process clear.

B.5.1 Engines

The top-level entity in any presentation is the *engine*. Parameters inside the engine point to a certain *window*, *camera*, *environment*, and a list of *actors* that should be processed by this engine. There are also parameters for controlling some high-level engine behaviors. Here is the list:

Parameter Specifier	Value Type	Default Value	Description
`vid-window`	`Address`	`Null`	The window for this engine to output to
`vid-environment`	`Address`	`Null`	The video environment to use in this engine
`vid-camera`	`Address`	`Null`	The camera to use in this engine
`vid-actors`	`AddrList`	`(AddrList)`	The actors to render in this engine (default is none)
`vid-clear`	`Bool`	`True`	Clear the plot buffer after building each frame?
`vid-clear-depth`	`Bool`	`True`	Clear the depth buffer after building each frame?
`vid-render-in-order`	`Bool`	`True`	Render actors in order given in list above?
`vid-output-descriptor`	`String`	`""`	A name for the output of this engine

Before you attempt to plot any frames from an engine, you must assign values to at least the first 4 parameters in the list above (window, environment, camera, and actors). The actors must be listed in an `AddrList`. In order for anything to show up in your output window, you must specify at least one actor.

In certain presentations, plot buffer clearing may not be necessary and can save a little time, or clearing may not be desired for artistic reasons. The `vid-clear` and `vid-clear-depth` parameters control whether the display and Z buffers are cleared before plotting each frame.

The `vid-render-in-order` parameter controls whether or not the actors are rendered in the same order they were placed in the list you gave for the `vid-actors` parameter. On certain platforms, you may be able to take advantage of parallel processing by setting this parameter to `False`, but in other situations, it may be necessary to render the actors in a certain order.

The `String` you specify for `vid-output-descriptor` will be used as a name for the output window if the delivery system supports named windows.

```
-> (vid-update myengine
               vid-window win1
               vid-environment env1
               vid-camera cam1
               vid-actors (AddrList act1 act2)
               vid-clear False
               vid-output-descriptor "Teresa's presentation")
<Null value>
->
```

B.5.2 Windows

A *window* entity specifies how the view obtained by the virtual *camera* will translate to a window on the output screen. There are only 3 parameters you can modify:

Parameter Specifier	Value Type	Default Value	Description
`win-location`	`Loc`	`(Loc 0 0)`	Location of *center* of window on a larger screen
`win-size`	`Size`	`(Size 512 512)`	Horizontal and vertical size of window in pixels
`win-brightness`	`Real`	`1.0`	Brightness of images in window (1.0 is normal)

The only tricky thing to remember here is that you give the location for the *center* of the window (not the upper left corner). Therefore, if you want your window to be perfectly centered on a larger screen, you simply give the pixel location of the center of the screen for the `win-location` parameter.

For example, to set window `win1` to a size of 320 horizontal by 200 vertical pixels and a position in the upper left corner of a screen, you would enter:

```
-> (vid-update win1
               win-size (Size 320 200)
               win-location (Loc 160 100))
<Null value>
->
```

To set the brightness to 150% of normal, you would enter:

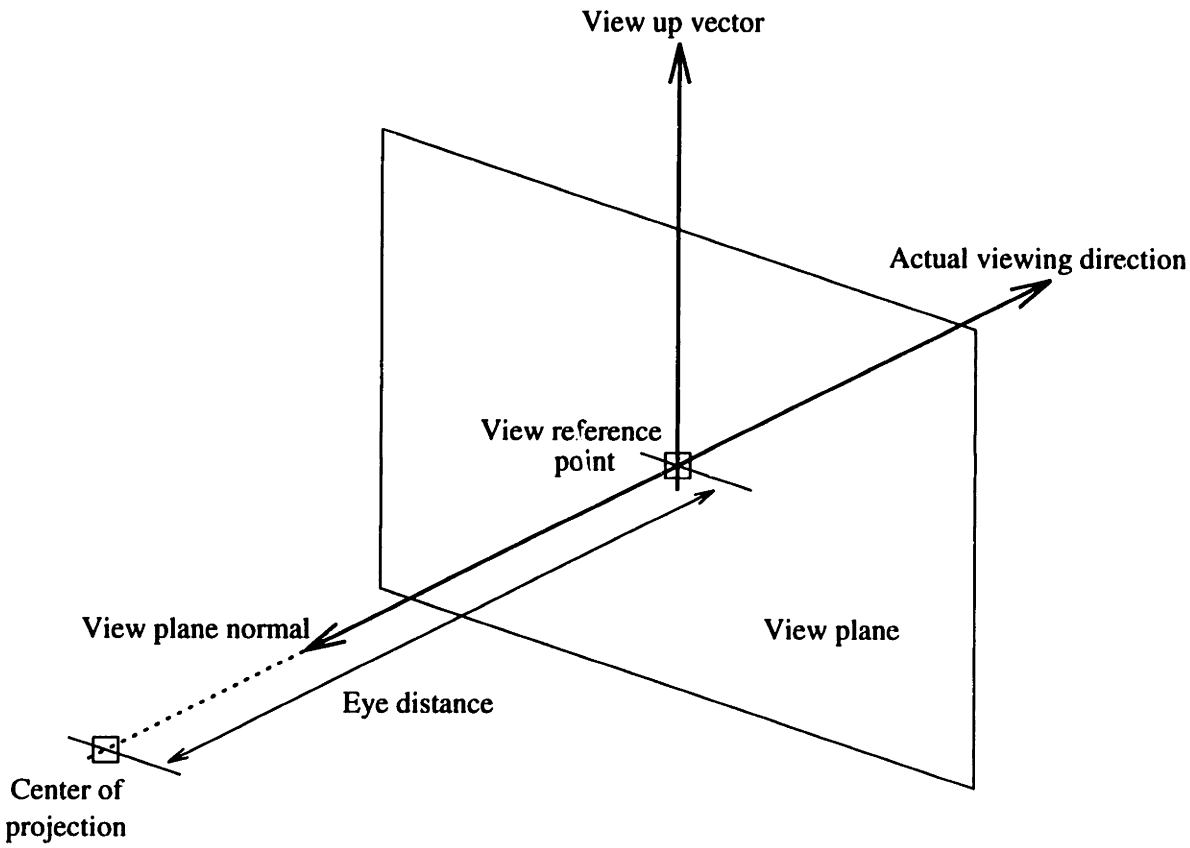
```
-> (vid-update win1 win-brightness 1.5)
<Null value>
->
```

B.5.3 Camera

The *camera* represents the details about a virtual "viewer" in a three dimensional space. The parameters function much like a camera in the computer graphics world:

Parameter Specifier	Value Type	Default Value	Description
`cam-ref-point`	`Pos`	`(Pos 0.0 0.0 0.0)`	View reference point
`cam-normal`	`Vec`	`(Vec 0.0 0.0 1.0)`	View plane normal (points *toward* the viewer)
`cam-up`	`Vec`	`(Vec 0.0 1.0 0.0)`	Up vector
`cam-eye-distance`	`Real`	`1.0`	Eye distance
`cam-viewport`	`Vec`	`(RealRect -1.0 -1.0 1.0 1.0)`	Viewport (rectangle on view plane) ordered: xmin, ymin, xmax, ymax

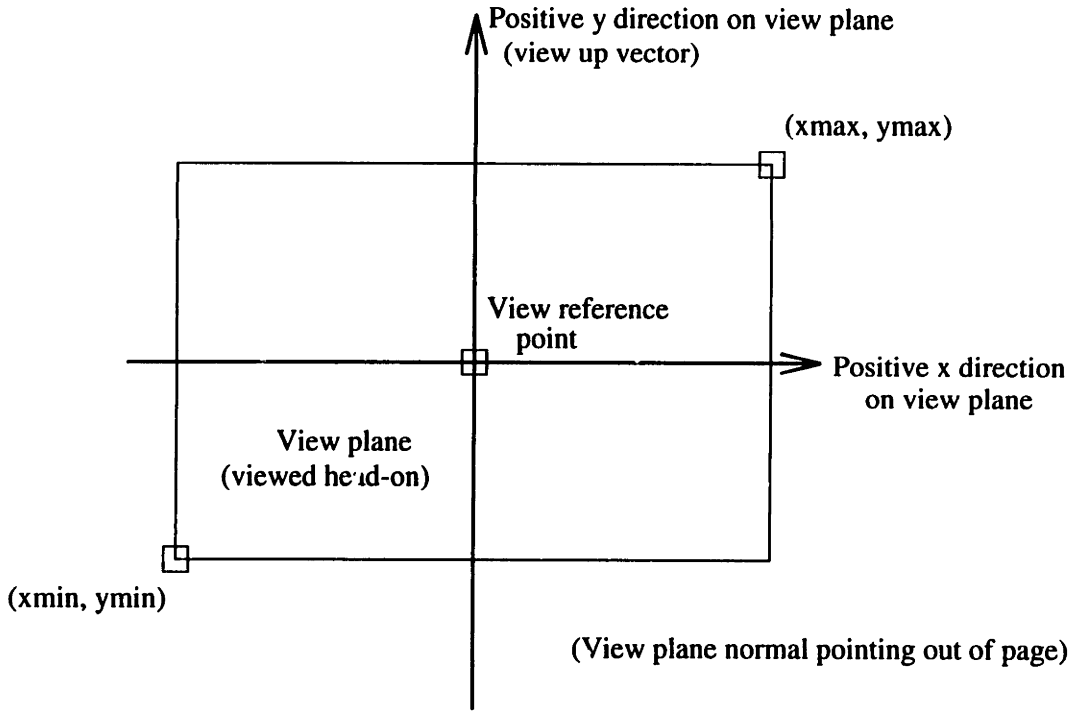
The *view reference point* and *view plane normal* determine a plane in space called the *view plane.* The *center of projection* is located *eye distance* units away from the view reference point in the direction of the view plane normal. (Note that this means the view plane normal points *toward* the viewer.)



An arbitrary point in the three-dimensional space is projected onto the view plane by finding the intersection of the view plane with the line that passes through both the center of projection and that point. The entire three-dimensional scene is mapped onto the two-dimensional view plane in this manner.

The portion of this view plane that is visible in your output window is called the *viewport.*

You first specify the direction to be considered as "up" as the *view up vector*. This vector points in the positive y direction on the view plane. Then, the viewport is specified by considering the view reference point to be the origin on the view plane and then giving a minimum and maximum x and y extent for the sides of the viewport.



All of the vectors and distances needed for the camera parameters should be specified in *world* units. Whether your world units are millimeters or miles depends on what units were used in your objects. If your objects are three-dimensional models, you will probably use the same units that were used when those objects were originally created. If your objects are two-dimensional images or movies, some file formats will allow you to specify a pixel-to-world scale factor for the images which will indicate the proper units to use. If no scale factor can be specified, then assume that the pixel size is the same as the world size (a 1:1 pixel-to-world scale factor).

The view plane normal and up-vector that you provide do not need to be normalized. All vectors are normalized and sanity-checked internally before any view calculations are performed.

The following example puts the view reference point at (100.0, 100.0, 100.0), and sets the view plane normal so that the camera is facing in the positive x direction and that "up" is considered to be in the positive y direction. The eye distance and viewport are set so that the viewport has a 90 degree horizontal field of view and a 45 degree vertical field of view

(a 2:1 "widescreen" aspect ratio).

```
-> (vid-update cam1
               cam-ref-point (Pos 100.0 100.0 100.0)
               cam-normal (Vec -1.0 0.0 0.0)
               cam-up (Vec 0.0 1.0 0.0)
               cam-eye-distance 256.0
               cam-viewport (RealRect -256.0 -128.0 256.0 128.0))
<Null value>
->
```

B.5.4 Environments

The video *environment* contains information about how the actors in the scene will be perceived by the camera. For instance, the environment would store information about the location and intensity of lights in a scene. However, in the current version of the structured video package, support for lights and other environment factors is not present. There are no parameters in the environment that can be modified. You can create an environment and assign it to a certain engine, but it has no effect on the appearance of the output (at least for now).

B.5.5 Actors

All of the other people and things in your production are the *actors*. Each actor has these parameters available to be modified:

Parameter Specifier	Value Type	Default Value	Description
`ac-object`	`Address`	`Null`	Video object
`ac-visibility`	`Real`	`1.0`	Opaqueness (from 0.0 to 1.0)
`ac-position-in-3d`	`Bool`	`True`	Position actor in 3 dimensions (3D mode)?
`ac-position`	`Pos`	`(Pos 0.0 0.0 0.0)`	3D position
`ac-rotation`	`Vec`	`(Vec 0.0 0.0 0.0)`	3D rotation vector
`ac-location`	`Loc`	`(Loc 0 0)`	2D location (ignored in 3D mode)
`ac-depth`	`Int`	`0`	Depth (the greater the closer) (ignored in 3D mode)
`ac-scale`	`Real`	`1.0`	Scale multiplier
`ac-frame`	`Int`	`0`	Temporal frame number
`ac-view`	`Int`	`0`	Spatial view number
`ac-use-depth`	`Bool`	`True`	Use depth buffer of the actor's object if available?
`ac-do-zbuffer`	`Bool`	`True`	Perform z-buffer operation on this actor?
`ac-use-alpha`	`Bool`	`True`	Use alpha buffer of the actor's object if available?
`ac-do-blending`	`Bool`	`True`	Perform alpha blending on this actor?

Each actor must point to a certain video *object* which holds all of the actual graphics data

to be used in rendering the actor. If you fail to set the `ac-object` parameter to a valid video object (created previously with `new-video-object`), the actor will never be rendered. `ac-visibility` controls how opaque the actor will be rendered—1.0 is fully opaque, 0.5 is half transparent, and 0.0 is invisible.

Each actor can be in one of two *positioning* modes (set with the `ac-position-in-3d` parameter). If the actor is in 3D mode, you can position and rotate the actor anywhere in the three-dimensional world (with the `ac-position` and `ac-rotation` parameters), and the actor will be scaled and rotated properly depending on the current camera pose when the frame is plotted. If the actor is the simpler 2D positioning mode, you can only place the actor at a certain pixel position in the current window (with the `ac-location` parameter)—the current camera and environment are ignored. The upper left corder of the window is considered the origin for 2D locations. In 2D mode, you can also scale the actor to different sizes using `ac-scale`, and if several actors overlap, you can set how they are layered using the `ac-depth` parameter (the greater the depth value the closer the object). In both positioning modes, when you provide the 3D position or the 2D location for the actor, the system places the *center* of the video object at that point.

`ac-frame` and `ac-view` determine which dataset of the video object will be rendered (if multiple datasets are available). Counting starts at 0. Frames are temporal and views are spatial. A traditional two-dimensional movie is just a series of temporal frames (with only 1 view), but it is possible to envision objects that could have both temporal and spatial components.

Some video objects will have depth map associated with each frame or dataset. Using the `ac-use-depth` and `ac-do-zbuffer` parameters, you can elect whether to make use of a depth buffer if it exists and whether or not to z-buffer the actor into the plotted scene (as opposed to simply overwriting the plot buffer with the rendered actor). Analogously, `ac-use-alpha` and `ac-do-blending` control whether to make use of an alpha channel if it exists, and if it does, whether or not to perform high-quality alpha blending (as opposed to treating the alpha channel as if it were binary).

```
-> (vid-update act1
               ac-object personobj
               ac-frame 10
               ac-position (Pos 50.0 60.0 70.0)
               ac-visibility 1.0)
<Null value>
->
```

B.6 Plotting frames

```
(realize-video video-engine)
```

Once you have assigned the desired values to all of the appropriate parameters inside each of the entities, you are ready to plot frames! You must have an engine that points to the correct window, camera, environment, and at least one actor. To actually plot the frame, simply call the `realize-video` function on your engine, and voila!

```
-> (realize-video myengine)
<Null value>
->
```

B.7 Handling time

```
(set-video-time time)
(get-video-time)
```

Most structured video presentations are nothing without time-varying output. Therefore, you need a way to keep track of the current time in order to plot the correct frame at the right time. The structured video system keeps an internal timer which runs constantly. Calling `get-video-time` returns the current time in seconds as a `Real` number. You can set the timer by calling `set-video-time` with any `Real` number. The internal time is immediately set to that value and continues running.

How you use the time to determine what settings to use for parameters is entirely up to you. Most scripts will want to take advantage of the `Timeline` data structure available in Isis. You could keep several timelines, each of which stores the value of a particular parameter over time. Then you need only obtain the current time and use it as an index into the timelines to find out what the parameters' values should be!

```
-> (set-video-time 10.0)
10.000000
-> (get-video-time)
14.015152
-> (get-video-time)
18.045455
-> (get-video-time)
22.196970
-> (get-video-time)
25.606061
->
```

B.8 Debugging

```
(video-info entity)
(set-video-verbose verbosity-number)
```

Two useful functions are available to help in debugging your presentation. Calling `video-info` on any structured video entity will print out extensive information about the state of that entity which may be useful in debugging. The `set-video-verbose` function accepts a single `Int` argument which will control the internal verbosity level of the machine-specific structured video plot routines for really low-level debugging.

B.9 A simple example

The following Isis script creates a structured video presentation consisting of two actors. Each actor points to its own video object containing 10 frames of image data each. The actors are placed in a three dimensional space and moved around in that space over time. One actor's frames play out at twice the speed of the other's. The presentation runs for 30 seconds and then exits.

```
# Example Structured Video script

# Flies 2 objects around in 3 dimensions and plays their frames
# at different speeds.  Total run time: 30 seconds.

(load "structured-av.isis")

(initialize-video-system)
```

```
(set movie1 (new-video-object "movie1.rgb" "Movie 1" "foreground1"))
(set movie2 (new-video-object "movie2.rgb" "Movie 2" "foreground2"))
(set myengine (new-video-engine))
(set win1 (new-window))
(set env1 (new-video-environment))
(set cam1 (new-camera))
(set act1 (new-actor))
(set act2 (new-actor))

(vid-update myengine
            vid-window win1
            vid-environment env1
            vid-camera cam1
            vid-actors (AddrList act1 act2)
            vid-output-descriptor "My first structured video presentation")

(vid-update cam1
            cam-eye-distance 100.0
            cam-viewport (RealRect -100.0 -100.0 100.0 100.0))

(vid-update act1 ac-object movie1)
(vid-update act2 ac-object movie2)

(set act1pos (new-timeline))
(key act1pos 0 (Pos 0.0 0.0 0.0))
(key act1pos 10 (Pos -100.0 -100.0 -100.0) linear)
(key act1pos 20 (Pos 100.0 100.0 -100.0) linear)
(key act1pos 30 (Pos 0.0 0.0 0.0) linear)

(set act2pos (new-timeline))
(key act2pos 0 (Pos -100.0 100.0 -100.0))
(key act2pos 5 (Pos 100.0 -100.0 0.0) linear)
(key act2pos 15 (Pos 100.0 100.0 -100.0) linear)
(key act2pos 25 (Pos -100.0 -100.0 0.0) linear)
(key act2pos 30 (Pos -100.0 100.0 -100.0) linear)

(start-video-system)

(set-video-time 0.0)

(set time (get-video-time))

(while (<= time 30.0)
       (begin
```

```
        (vid-update act1
                    ac-position (act1pos time)
                    ac-frame (% (* 10.0 time) 10))
        (vid-update act2
                    ac-position (act2pos time)
                    ac-frame (% (* 20.0 time) 10))
        (realize-video myengine)
        (set time (get-video-time))))

(stop-video-system)
(terminate-video-system)
```

B.10 Making your presentation interactive

Let's face it—structured video presentations could be much more fun and interesting if they were interactive. The document *User Interfaces with Isis* introduces a package of functions for creating user interfaces. Using those functions, you can set up a standard interface for obtaining user input of different types from many different sources, and then use those input values in your script to control various aspects of the production.

B.11 Final remarks

Although the structured video package by itself is completely platform independent, the individual delivery systems on various platforms will differ highly in the way they render your presentation. Some delivery systems do not have the capability to perform alpha blending or control window brightness. Some parameters in some entities may be completely ignored. There should be a document available describing any wierdnesses of each system that you should read before authoring a serious presentation.

Appendix C

Structured Audio with Isis

C.1 Introduction

This document will explain how to create structured audio presentations using the *Isis* scripting language. You should first read the document *Isis: A multi-purpose interpretive scripting language* which explains the details of the Isis language itself.

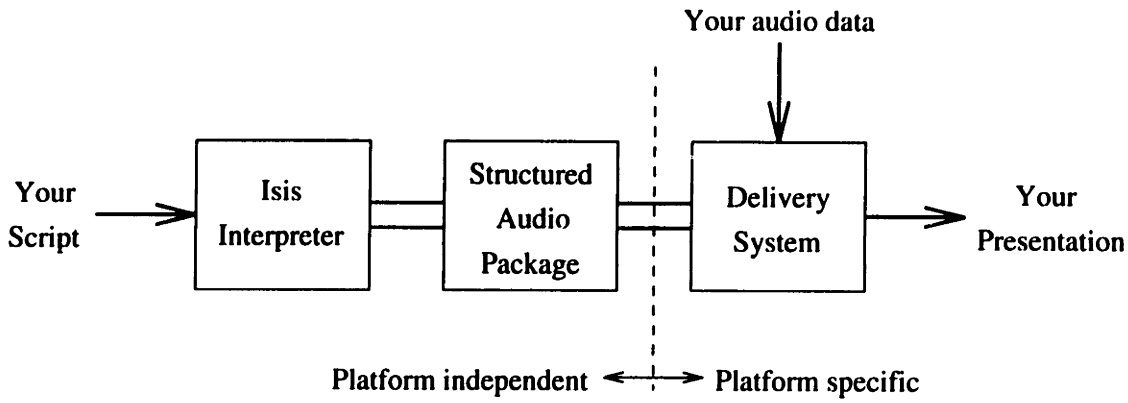
The term *structured audio* refers to a representation for audio in terms of *objects*, each of which is a piece of audio. Several component objects are manipulated and assembled in some manner over time to produce a full audio presentation. Representing audio in this way tends to greatly simplify the problems of interactive and personalized delivery.

The structured audio package lets you create a three-dimensional virtual environment for the playback of sounds. You can place any number of sounds and a single listener anywhere in a virtual space and modify playback and acoustical characteristics in order to achieve a desired effect. The results are rendered on a set of speakers surrounding the real listener in the physical area where the presentation will take place.

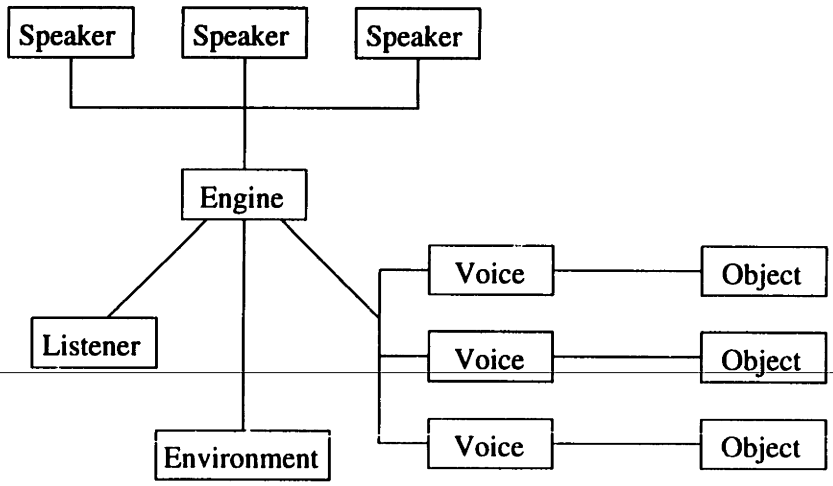
The package of Isis functions described in this document will allow you to express highly interactive structured audio presentations in a platform-independent manner. Since Isis is an interpretive scripting language, you can listen to and change aspects of your presentation instantly without having to recompile entire programs each time you make a small modification. You will be able to use delivery systems on several different platforms to listen to and interact with your production. Each platform will offer advantages over the others. For example, Isis and the structured audio package running on an Alpha processor will provide real-time CD-quality directional playback with special effects, but with only

a limited number of voices. Other platforms may offer more voices but a reduced sound quality.

Let's begin with a general overview of the design of the system. There are 3 main segments. At the highest level, there is the platform-independent Isis script interpreter. The structured audio package is then accessed in Isis merely by calling various built-in Isis functions. These functions are also machine-independent, so it is possible for you to write a single script that will run on any platform. It is then the job of the structured audio package to use machine specific routines and libraries to deliver your presentation as effectively as possible on your platform.



The structured audio package lets you create and manipulate six different kinds of abstract entities: objects, engines, voices, listeners, environments, and speakers.



Essentially, consider yourself as the *listener*, the other people and things that make noise in the production as the *voices*, and the details of how you perceive those people and things

as the *environment*. The *speakers* describe how what the listener hears will be translated to the speakers in your room. The *engine* can loosely be thought of as the *translator* from the virtual world of the presentation to the output devices in the real world. Using the information stored in a certain listener, an environment, one or more voices, and one or more speakers, the engine does all the work to build the mixed audio for presentation. You can create more than one engine in a single session, each of which can share voices, listeners, etc., with any of the other engines. This gives you the power to control several presentations using many different sets of speakers simultaneously.

Each voice must point to an *object*, which is an entity that refers to actual audio data, such as a sound effect or a piece of music. More than one voice can use the same object simultaneously, but each voice can only point to a single object at a time. Only raw audio data (or information about where it is located) is stored in the *object*. Information about how that data gets rendered is part of the *voice*.

The functions available to you in Isis allow you to create instances of these six kinds of entities and modify parameters inside each of them. A typical script will begin by calling initialization functions and creating all the needed entities. To output the actual audio, the script need only set the appropriate parameters inside each entity to the desired values (possibly taking into account user input) and then call a function that renders the audio with those settings. The audio plays continuously, but if this "update and render" routine is performed repeatedly, it is possible to achieve essentially real-time feedback for changes! The following sections of this manual will explain this process in greater detail.

C.2 Running Isis and the Structured Audio Package

In order to use the functions in the structured audio package, you first must check that the version of Isis you are using is linked to the package. When you start Isis, you will see something like this:

```
*** Isis (version 2.0)
*** by Stefan Agamanolis
+++ Structured Video Package
=== Structured Audio Package
--- User Interface Package

->
```

If you don't see "`Structured Audio Package`" listed here, then you must use a different version of Isis. Ask your friendly Isis software maintainer where to find the correct version.

Next, you must load the file `structured-av.isis` which should be located in the same place as the Isis interpreter program, or in a directory where standard Isis scripts are stored. Although not required, this file contains helpful type definitions that will simplify using the structured audio system. Make sure to specify the entire pathname if necessary. Otherwise, if the file `structured-av.isis` is in your current directory, you can simply type:

```
-> (load "structured-av.isis")
Reading from script 'structured-av.isis'...
13 items processed
<Null value>
->
```

For reference, this file makes the following type definitions:

```
# structured-av.isis
# Helpful type definitions structured audio and video scripts

(newtype (Pos Real Real Real))
(newtype (Vec Real Real Real))
(newtype (Dim Real Real Real))

(newtype (Loc Int Int))
(newtype (Size Int Int))

(newtype (Vec4 Real Real Real Real))
(newtype (Matrix4 Vec4 Vec4 Vec4 Vec4))  # each vector is a ROW

(newtype (Rect Int Int Int Int))
(newtype (RealRect Real Real Real Real))

(newtype (View Vec Vec Vec Real RealRect))

(newtype (Poslist Pos ...))
(newtype (Viewlist View ...))
```

C.3 Initializing and controlling system state

```
(initialize-audio-system)
(start-audio-system)
(stop-audio-system)
(reset-audio-system)
(terminate-audio-system)
```

The first package function you must call in any script that uses structured audio is `initialize-audio-system`. This function should only ever be called once. Then, before outputting any audio from any engine, you must call `start-audio-system` which will switch on any speakers that you have created and prepare them for output. `stop-audio-system` will turn off any speakers in use—but they can be turned on again with another call to `start-audio-system`. `reset-audio-system` allows you to re-initialize the system to the same state as when the system was first initialized. `terminate-audio-system` will stop all audio output, turn off all speakers, and shut down any external processes associated with the delivery system. It should only ever be called once, usually at the very end of a script.

```
-> (initialize-audio-system)
<Null value>
-> (start-audio-system)
<Null value>
->
```

C.4 Creating structured audio entities

```
(new-audio-engine)
(new-speaker)
(new-listener)
(new-audio-environment)
(new-voice)
(new-audio-object data-filename internal-name)
```

These functions will create new structured audio entities. Each returns a value of type `Address` which you will use to refer to the entity later, so you should always be sure to store it (usually by assigning it to a variable).

```
-> (set myengine (new-audio-engine))
0x402d4000
-> (set spkr1 (new-speaker))
0x402d4020
-> (set env1 (new-audio-environment))
0x402d4040
-> (set listener1 (new-listener))
0x402d4060
-> (set voice1 (new-voice))
0x402d4080
->
```

Two `String` arguments are required by `new-audio-object`. First, `data-filename` should

be the filename or URL where the audio data may be found. URL support may or may not be present on your system. The format of the data is inferred from its filename or extension. Second, the **internal-name** can be whatever you like as it is only used in printing debugging information about the object, not in any processing.

```
-> (set wind (new-audio-object "/grad/stefan/wind.raw" "Wind"))
0x402d40a0
-> (set speech (new-audio-object "/grad/stefan/speech.raw" "Speech"))
0x402d40c0
-> (set piano (new-audio-object "/grad/stefan/piano.aiff" "Piano"))
0x402d40e0
->
```

C.5 Updating parameters

```
(aud-update entity parameter value parameter value ...)
```

There are several parameters inside each structured audio entity, each of which has a value that you may change to alter the state of that entity in some way. The parameters are given default values when the entity is first created. It is your job to set these parameters appropriately before calling the function that renders the audio. The way you set these parameters is by calling the **aud-update** function. This function accepts a variable number of arguments. The first argument must be the entity you wish to set parameters within. Then you pass a parameter specifier followed by the value you wish to give that parameter. You may pass as many parameters and corresponding values in a single call as you like. The examples in the following sections should make this process clear.

C.5.1 Engines

The top-level entity in any presentation is the *engine*. Parameters inside the engine point to a list of *speakers*, a *listener*, an *environment*, and a list of *voices* that should be processed by the engine. There are also parameters for controlling some high-level engine states. Here is the list:

Parameter Specifier	Value Type	Default Value	Description
aud-speakers	**AddrList**	**(AddrList)**	The speakers for this engine to output to
aud-environment	**Address**	**Null**	The audio environment to use in this engine
aud-listener	**Address**	**Null**	The listener to use in this engine
aud-voices	**AddrList**	**(AddrList)**	The voices to render in this engine (default is none)
aud-output-descriptor	**String**	**""**	A name for the output of this engine

Before you attempt to output any audio from an engine, you must assign values to at least the first 4 parameters in the list above (speakers, environment, listener, and voices). The speakers and voices must be listed in values of type **AddrList**. In order for anything to be heard, you must specify at least one voice and one speaker.

The **String** you specify for **aud-output-descriptor** will be used as a name for the output from this engine if the delivery system supports a naming scheme of some kind.

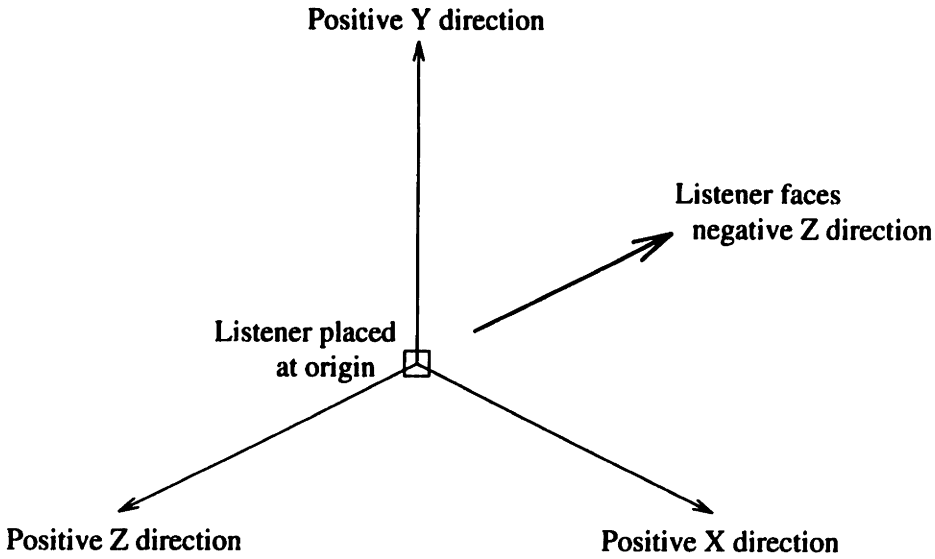
```
-> (aud-update myengine
                aud-speakers (AddrList spkr1 spkr2)
                aud-environment env1
                aud-listener listener1
                aud-voices (AddrList voice1 voice2)
                aud-output-descriptor "Amy's presentation")
<Null value>
->
```

C.5.2 Speakers

A *speaker* entity specifies where the physical speakers are placed with respect to the listener in your presentation room. This information is needed in order to render the correct directional mix of the audio for each speaker. There is only one parameter you can modify:

Parameter Specifier	Value Type	Default Value	Description
sp-location	Pos	(Pos 0.0 0.0 -1.0)	Location of speaker in physical presentation space

The position that you provide here must be given in a coordinate space where the physical listener is at the origin and is facing in the negative Z direction. The positive X axis points to the right of the listener, the positive Y axis points upward, and the positive Z axis points backwards (behind the listener). Positions must be given using *meters* as units.



For example, if you have a double speaker system where both speakers are 1 meter in front of the listener and one is 1 meter to the left and the other is 1 meter to the right, you would enter:

```
-> (aud-update spkr1 sp-position (Pos -1.0 0.0 -1.0))
<Null value>
-> (aud-update spkr2 sp-position (Pos 1.0 0.0 -1.0))
<Null value>
->
```

C.5.3 Listener

The *listener* represents the details about a virtual "hearing" entity in a three dimensional space. The parameters function much like an extremely simple camera in the computer graphics world:

Parameter Specifier	Value Type	Default Value	Description
`li-position`	`Pos`	`(Pos 0.0 0.0 0.0)`	Listener position
`li-normal`	`Vec`	`(Vec 0.0 0.0 1.0)`	Listener normal (points *toward* the listener)
`li-up`	`Vec`	`(Vec 0.0 1.0 0.0)`	Listener up vector

The virtual listener's position is given for `li-position`, and the *opposite* of the direction she is facing is given for `li-normal`. The direction that is considered "up" for the listener is given for `li-up`. `li-normal` and `li-up` do *not* have to be specified as unit vectors.

Consider that you are positioning the listener in a rectangular room with the origin at the exact center of the room. *Meters* must be used as the units for these positions unless you have altered the **ae-dist-scale** parameter in the *environment* entity. The size and acoustical properties of the room can also be modified in the environment. Be sure not to position the listener outside of the room as specified in the current environment.

The following example places the listener at (1.0, 1.0, 1.0) and faces her in the positive X direction. "Up" is set to point in the positive Y direction.

```
-> (aud-update listener1
               li-position (Pos 1.0 1.0 1.0)
               li-normal (Vec -1.0 0.0 0.0)
               li-up (Vec 0.0 1.0 0.0))
<Null value>
->
```

C.5.4 Environments

The audio *environment* contains information about how the voices in the scene will be perceived by the listener. For instance, the environment stores information about the size and acoustical properties of the virtual rectangular "room" where the action is taking place. The reflection coefficients for the walls and the amount of reverberation in the room can be modified. Here are the parameters:

Parameter Specifier	Value Type	Default Value	Description
ae-dimensions	Dim	(Dim 10.0 10.0 10.0)	X Y Z dimensions of rectangular room
ae-size-scale	Real	1.0	Multiplier for room dimensions
ae-reflection-coefs	Real	(RealList 0.5 0.5 0.5 0.5 0.5 0.5)	Reflection coefficients of six walls
ae-max-reflections	Int	5	Max number of reflections for bounced sounds
ae-reverb-scale	Real	0.0	Reverb (0.0 to 1.0)
ae-dist-scale	Real	1.0	Multiplier for positions and distances

The dimensions of the room in the X, Y, and Z directions are given in *meters* for the **ae-dimensions** parameter. The value of **ae-size-scale** is multiplied with each dimension, so you can easily increase or decrease the size of the room proportionally or use different units for the dimensions.

The reflection coefficients are specified for the walls corresponding to each of the six coordinate axis directions, in this order: positive X, negative X, positive Y, negative Y, positive Z, negative Z. Each coefficient should be between 0 and 1. The listener will only hear virtual sound sources that have reflected off the walls less than **ae-max-reflections** times. Setting this value very high may have a detrimental effect on the overall performance of the audio

delivery system since many more bounced sources of a sound will need to be processed.

The amount of reverb in the room is controlled by giving **ae-reverb-scale** a value between 0 and 1.

Normally, all voice and listener positions must be in *meters* relative to the exact center of the room. However, **ae-dist-scale** will be multiplied to all positions, thereby providing an easy way to use different units if you wish. For example, if you wanted to position everything in millimeters rather than meters, you would set this parameter to 1000. This multiplier only affects positions, not room dimensions (which is controlled by **ae-size-scale**).

Since the rectangular room is symmetrical about all three axes, it does not really matter which axis you choose to represent which directions in your virtual space, as long as your positions and dimensions are consistent with whatever system you choose. However, by convention, usually the Y axis is considered the up-and-down axis.

```
-> (aud-update env1
               ae-dimensions (Dim 20.0 10.0 20.0)
               ae-reflection-coefs (RealList 0.5 0.5 0.3 0.3 0.7 0.7)
               ae-max-reflections 5
               ae-reverb-scale 0.5)
<Null value>
->
```

C.5.5 Voices

All of the noises emitted by the people and things in your production are modeled by the *voices*. Each voice has these parameters available to be modified:

Parameter Specifier	Value Type	Default Value	Description
vo-object	**Address**	**Null**	Audio object
vo-start-time	**Real**	**-1.0**	Start time (-1.0 for immediately)
vo-end-time	**Real**	**-1.0**	End time (-1.0 for immediately, overrides start time)
vo-position	**Pos**	**(Pos 0.0 0.0 -1.0)**	3D Position
vo-max-depth	**Real**	**100.0**	Maximum distance of bounced sources from listener
vo-volume	**Real**	**0.0**	Volume in dB (0.0 for natural volume)
vo-loop	**Bool**	**False**	Play the sound in a continuous loop?

Each voice must point to a certain audio *object* which holds all of the actual audio data to be used in rendering the voice. If you fail to set the **vo-object** parameter to a valid audio object (created previously with **new-audio-object**), the voice will never be rendered.

You can indicate very specific start and ends time for the sound by setting **vo-start-time** and **vo-end-time** accordingly. The sound will end at the end time or at the end of the sound recording, whichever happens first. The current time can be set or checked with the **set-audio-time** and **get-audio-time** functions described later. Otherwise, if you simply want the sound to start immediately and play forever, set **vo-start-time** to **-1.0** and set **vo-end-time** to something very large. The sound will start playing as soon as you call **realize-audio** (described later). If you want to stop a sound that is playing immediately or prevent a voice from being heard, set **vo-end-time** to **-1.0**.

You provide the three-dimensional position of the voice for the 0**vo-position** parameter. This position must be within the dimensions of the rectangular room described in the current environment. Note that the position must be specified in *meters* unless you have altered the **ae-dist-scale** parameter in the environment. The **vo-max-depth** parameter determines a maximum distance for the sound or virtual "bounced" sources of the sound—sources above this distance will not be rendered. Setting this value higher increases the amount of processing necessary to play the audio since more bounced sources of sounds will need to be rendered.

vo-volume controls the loudness the voice. Values should be expressed in decibels.

Setting **vo-loop** to **True** will cause the voice's sound object to be played over and over again until the end time is reached or until some other action interrupts the voice. Loop mode is useful for background sounds that need to be played continuously and repeatedly in the background.

```
-> (aud-update voice1
               vo-object wind
               vo-start-time -1.0
               vo-end-time 100000.0
               vo-position (Pos 8.0 0.0 8.0)
               vo-volume -10.0
               vo-loop True)
<Null value>
-> (aud-update voice2
               vo-object speech
               vo-start-time 10.0
               vo-end-time 25.0
               vo-position (Pos 1.0 1.0 -1.0)
               vo-volume 5.0)
<Null value>
->
```

C.6 Rendering the audio

```
(realize-audio audio-engine)
```

Once you have assigned the desired values to all of the appropriate parameters inside each of the entities, you are ready to render the audio! You must have an engine that points to the correct speakers, listener, environment, and at least one voice. To actually output the audio, simply call the `realize-audio` function on your engine, and voila! All of the settings you have made in the entities associated with that engine will take effect immediately. If you changed one of the objects that a voice was pointing to, the new object will start playing from its beginning. Otherwise, you can make any other changes to the settings of a voice and the audio object that is playing will continue uninterrupted.

```
-> (realize-audio myengine)
<Null value>
->
```

C.7 Handling time

```
(set-audio-time time)
(get-audio-time)
```

The structured audio package provides the capability to start and stop sounds at exact times. You may also want to change the audio presentation in some way depending on the value of a timer. The structured audio system keeps an internal timer which runs constantly. Calling `get-audio-time` returns the current time in seconds as a `Real` number. You can set the timer by calling `set-audio-time` with any `Real` number. The internal time is immediately set to that value and continues running.

How you use the time to determine what settings to use for parameters is entirely up to you. Most scripts will want to take advantage of the `Timeline` data structure available in Isis. You could keep several timelines, each of which stores the value of a particular parameter over time. Then you need only obtain the current time and use it as an index into the timelines to find out what the parameters' values should be!

```
-> (set-audio-time 10.0)
10.000000
-> (get-audio-time)
14.015152
-> (get-audio-time)
18.045455
-> (get-audio-time)
22.196970
-> (get-audio-time)
25.606061
->
```

C.8 Debugging

```
(audio-info entity)
(set-audio-verbose verbosity-number)
```

Two useful functions are available to help in debugging your presentation. Calling **audio-info** on any structured audio entity will print out extensive information about the state of that entity which may be useful in debugging. The **set-audio-verbose** function accepts a single **Int** argument which will control the internal verbosity level of the machine-specific structured audio rendering routines for really low-level debugging.

C.9 A simple example

The following Isis script creates a structured audio presentation consisting of two voices. Each voice points to a separate audio object. The voices are placed in a three dimensional space and moved around over time. The acoustics of the room are also modified while the presentation runs.

```
# Example Structured Audio script
# Flies 2 voices around a listener 5 times.

(load "structured-av.isis")

(initialize-audio-system)

(set music (new-audio-object "music.raw" "Music"))
(set speech (new-audio-object "speech.raw" "Speech"))
```

```
(set myengine (new-audio-engine))
(set spkr1 (new-speaker))
(set spkr2 (new-speaker))
(set env1 (new-audio-environment))
(set listener1 (new-listener))
(set voice1 (new-voice))
(set voice2 (new-voice))

(aud-update myengine
            aud-speakers (AddrList spkr1 spkr2)
            aud-environment env1
            aud-listener listener1
            aud-voices (AddrList voice1 voice2)
            aud-output-descriptor "My first structured audio presentation")

(aud-update spkr1 sp-position (Pos -1.0 0.0 -1.0))
(aud-update spkr2 sp-position (Pos 1.0 0.0 -1.0))

(aud-update env1 ae-reflection-coefs (RealList 0.5 0.5 0.3 0.3 0.7 0.7))

(aud-update voice1
            vo-object music
            vo-end-time 100000.0
            vo-loop True)

(aud-update voice2
            vo-object speech
            vo-end-time 100000.0
            vo-loop True)

(set voice1pos (new-timeline))
(key voice1pos 0 (Pos 1.0 0.0 1.0))
(key voice1pos 5 (Pos -1.0 0.0 1.0) linear)
(key voice1pos 10 (Pos -1.0 0.0 -1.0) linear)
(key voice1pos 15 (Pos 1.0 0.0 -1.0) linear)
(key voice1pos 20 (Pos 1.0 0.0 1.0) linear)

(set voice2pos (new-timeline))
(key voice2pos 0 (Pos -1.0 0.0 -1.0))
(key voice2pos 5 (Pos -1.0 0.0 1.0) linear)
(key voice2pos 10 (Pos 1.0 0.0 1.0) linear)
(key voice2pos 15 (Pos 1.0 0.0 -1.0) linear)
(key voice2pos 20 (Pos -1.0 0.0 -1.0) linear)

(set revval (new-timeline))
```

```
(key revval 0 0.0)
(key revval 10 1.0 linear)
(key revval 20 0.0 linear)

(start-audio-system)

(set-audio-time 0.0)

(set time (get-audio-time))

(while (<= (get-audio-time) 100.0)
       (begin
         (aud-update voice1 vo-position (voice1pos time))
         (aud-update voice2 vo-position (voice2pos time))
         (aud-update env1 ae-reverb-scale (revval time))
         (realize-audio myengine)
         (set time (% (get-audio-time) 20.0))))

(stop-audio-system)
(terminate-audio-system)
```

C.10 Making your presentation interactive

Let's face it—structured audio presentations could be much more fun and interesting if they were interactive. The document *User Interfaces with Isis* introduces a package of functions for creating user interfaces. Using those functions, you can set up a standard interface for obtaining user input of different types from many different sources, and then use those input values in your script to control various aspects of the production.

C.11 Final remarks

Although the structured audio package by itself is completely platform independent, the individual delivery systems on various platforms will differ highly in the way they render your presentation. Some delivery systems may be able to position voices in only one- or two-dimensional virtual worlds, and others may not be able to handle more than two speakers. Some parameters in some entities may even be completely ignored. There should be a document available describing any wierdnesses of each system that you should read before authoring a serious presentation.

Appendix D

User Interfaces with Isis

D.1 Introduction

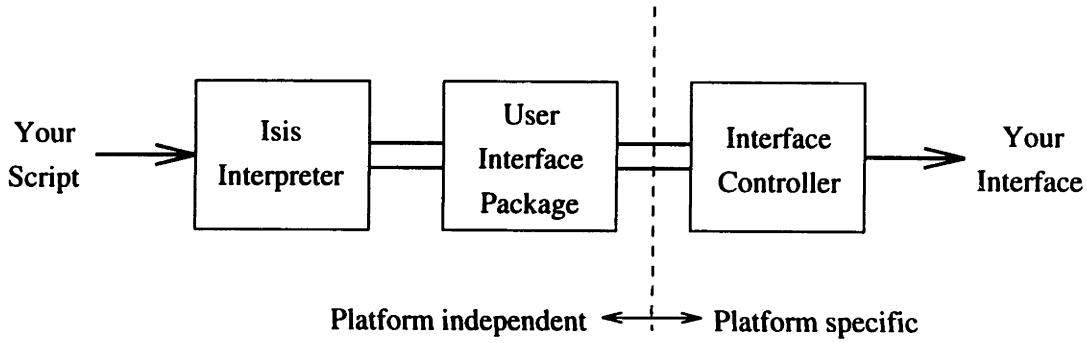
This document will explain how to create user interfaces with the *Isis* scripting language. You should first read the document *Isis: A multi-purpose interpretive scripting language* which explains the details of the Isis language itself.

The structured audio and video packages for Isis allow you to author complex presentations that incorporate a wide variety of media forms. However, presentations of this kind in many cases could be much more compelling if they allowed the viewer or listener to interact in some way. Some kind of interface into Isis would be needed to provide input that could be used to alter the media delivery in some way.

The user interface package for Isis enables you to create a standard user interface that will work on any delivery platform, from common workstations and operating systems to experimental audio and video processing hardware. The problem with designing a standard user interface system is that not all machines will offer the same input possibilities as the others. Some systems may manage a windowing system that supports buttons and scrollbars, but many times, especially with experimental machines, none of these conveniences are present and different and sometimes odd input devices must be used instead.

The Isis interface package solves this problem by making the least number of assumptions possible about the specific capabilities of various platforms. Windows, buttons, scrollbars, menus, etc., are all assumed to be nonexistent. Event processing is also assumed not to be present. The operation of the package is governed by the simplest possible model of interaction.

Let's begin with a general overview of the design of the system. There are 3 main segments. At the highest level, there is the platform-independent Isis script interpreter. The user interface package is then accessed in Isis merely by calling various built-in Isis functions. These functions are also machine-independent, so it is possible for you to write a single script that will run on any platform. It is then the job of the interface package to connect to a machine-specific interface controller that will realize your interface in the most effective way possible for your system.



The user interface package lets you create and manipulate two different kinds of entities: *interfaces* and *inputs*. An *interface* is nothing more than a collection of related *inputs*. An *input* is an abstract notion of a polled "input device" which can receive data of only 5 different kinds: boolean, integer number, real number, character, or string. Each of these 5 input kinds corresponds to a particular Isis primitive data type: `Bool`, `Int`, `Real`, `Char`, and `String`. Each input device can be configured in different ways to limit the possible range of acquired values.

Each *input* that is created and configured in a script is tied to an input device on the specific platform on which Isis is running. How that input device is realized depends on what input possibilities are available on that platform. For example, an integer input configured to have certain maximum and minimum value might be rendered as a scroll bar inside some kind of graphical user interface, or, on more primitive systems, it might be tied to a certain knob in a physical knob box connected to the computer.

A typical script needing a user interface will begin by calling initialization functions and creating and configuring all the needed interfaces and inputs. To obtain input values, the script need only call a function that checks the value of a particular *input* entity. The following sections of this manual will explain this process in greater detail.

D.2 Running Isis and the User Interface package

In order to use the functions in the user interface package, you first must check that the version of Isis you are using is linked to the package. When you start Isis, you will see something like this:

```
*** Isis (version 2.0)
*** by Stefan Agamanolis
+++ Structured Video Package
=== Structured Audio Package
--- User Interface Package

->
```

If you don't see "`User Interface Package`" listed here, then you must use a different version of Isis. Ask your friendly Isis software maintainer where to find the correct version.

D.3 Initializing

```
(initialize-interface)
```

The first package function you must call in any script that creates user interfaces is `initialize-interface`. This function should only ever be called once (with no arguments) near the beginning of your script.

```
-> (initialize-interface)
<Null value>
->
```

D.4 Creating and manipulating interfaces

```
(create-interface name)
(start-interface interface)
(update-interface interface)
(stop-interface interface)
(destroy-interface interface)
```

Each *interface* that you create will contain a collection of *inputs*. How you organize which

inputs belong to which interfaces is your decision. You may want to put all of your inputs in a single interface, or you may want to split them up among several interfaces.

The `create-interface` function creates a new interface entity. You must provide a `String` name which will be used as a title to your interface if some kind of naming system is available on your system. This function returns a value of type `Address` which you will use to refer to the interface later, so you should always be sure to store it (usually by assigning it to a variable).

After you have added the desired inputs to each interface you create (described in the next section), you can actually turn the interface on and off by calling `start-interface` and `stop-interface` on it.

Calling `update-interface` will cause the system to grab all of the current values of the input devices. This allows you to grab the entire interface state at a particular instant in time rather than poll each input separately over a longer period. You will have to call this function whenever you want the current condition of input devices in the interface. Between each call, the state of the interface and its inputs is constant.

When you are completely done with a particular user interface and never want to use it again, you can call `destroy-interface` on it.

```
-> (initialize-interface)
<Null value>
-> (set myinterface (create-interface "Aimee's first interface"))
0x303dfc80
-> (start-interface myinterface)
<Null value>
-> (update-interface myinterface)
<Null value>
-> (stop-interface myinterface)
<Null value>
-> (destroy-interface myinterface)
<Null value>
->
```

D.5 Creating and manipulating inputs

```
(add-input interface input-name input-type)
(configure-input input minimum maximum step)
(input-value input)
(set-input-value input value)
```

Once you have created an interface, you may add input devices to it using the `add-input` function. You must pass three arguments: the interface you wish the input to be a part of, a `String` name for the input, and the data type of input. The data type can be one of the following 5 Isis primitive data types: `Bool`, `Int`, `Real`, `Char`, or `String`. A value of type `Address` is returned which you will use to refer to this input in the future, so be sure to store it.

```
-> (set intinput (add-input myinterface "An Integer" Int))
0x303dfcb0
-> (set realinput (add-input myinterface "A Real" Real))
0x303dfcf0
-> (set boolinput (add-input myinterface "A Boolean" Bool))
0x303dfd40
-> (set charinput (add-input myinterface "A Character" Char))
0x303dfd80
-> (set stringinput (add-input myinterface "A String" String))
0x303dfdc0
->
```

4 of the 5 kinds of inputs can be configured to have a maximum, minimum, and step value (if applicable). The first argument to the `configure-input` function should be the input device, and then this function will accept a variable number of additional arguments depending on what kind of input is being configured.

- `Int` and `Real` inputs can be configured by passing 3 additional `Int`s or `Real`s to `configure-input` in this order: minimum value, maximum value, and step value. The value of the input device will be constrained to the range defined by the maximum and minimum value with the granularity specified by the step value.
- `Char` inputs are configured by passing only 2 additional arguments—the minimum and maximum characters (in ASCII space). No step value may be specified.
- The minimum and maximum lengths for `String` inputs is configured by passing 2 arguments—the minimum length and the maximum length for the `String` value.
- `Bool` inputs cannot be configured since there are only 2 possible return values—`True` and `False`.

```
-> (configure-input intinput 0 100 1 0)
<Null value>
-> (configure-input realinput 0.0 100.0 1.0 0.0)
<Null value>
-> (configure-input charinput 'a' 'z')
<Null value>
-> (configure-input stringinput 0 20)
<Null value>
->
```

Once the input has been created and configured to your liking, you can get or set the value of the input device using the **input-value** and **set-input-value** functions. **input-value** takes one argument—the input device you want to check. **set-input-value** accepts the input device and its new value as its arguments.

You must remember to call **update-interface** in order to grab the most recent state of the input devices before making calls to **input-value**.

The following example shows what you might see before and after the input devices in an interface have been modified.

```
-> (update-interface myinterface)
<Null value>
-> (input-value intinput)
0
-> (input-value realinput)
0.000000
-> (input-value charinput)
'a'
-> (input-value boolinput)
False
-> (input-value stringinput)
""
-> (update-interface myinterface)
<Null value>
-> (input-value intinput)
58
-> (input-value realinput)
30.000000
-> (input-value charinput)
'q'
-> (input-value boolinput)
True
-> (input-value stringinput)
"Idiot"
```

D.6 Debugging

```
(set-interface-verbose verbosity-number)
```

One useful function is available to help in debugging your interface. The `set-interface-verbose` function accepts a single `Int` argument which will control the internal verbosity level of the machine-specific interface controller routines—useful for watching what's going on inside the system.

D.7 A simple example

The following Isis script creates a simple user interface and enters a loop that does a mathematical operation on the current values of the inputs. The input values and the result of the calculation are displayed continuously.

```
# Example User Interface script

# Creates simple interface and does a mathematical operation on the values

(initialize-interface)

(set myinterface (create-interface "Pamela's first interface"))

(set int1 (add-input myinterface "Integer 1" Int))
(set int2 (add-input myinterface "Integer 2" Int))
(set real1 (add-input myinterface "Real 1" Real))
(set real2 (add-input myinterface "Real 2" Real))
(set bool1 (add-input myinterface "Sign" Bool))
(set bool2 (add-input myinterface "Exit" Bool))

(configure-input int1 0 100 1)
(configure-input int2 0 100 1)
(configure-input real1 0.0 20.0 0.1)
(configure-input real2 -100.0 -50.0 0.01)

(set-input-value int1 42)
(set-input-value int2 69)
(set-input-value real1 13.0)
(set-input-value real2 -69.42)
(set-input-value bool1 True)
```

```
(set-input-value bool2 False)

(start-interface myinterface)

(while (not (input-value bool2))
       (begin
         (update-interface myinterface)
         (set i1 (input-value int1))
         (set i2 (input-value int2))
         (set r1 (input-value real1))
         (set r2 (input-value real2))
         (set op (if (input-value bool1) + -))
         (set result (op (op r1 r2) (op i1 i2)))
         (display i1 i2 r1 r2 result)))
```

D.8 Final remarks

Although the user interface package by itself is completely platform-independent, the controllers on various platforms will differ highly in the way they realize your interface. Some systems may be not handle more than one interface or may not be able to acquire certain kinds of values. There should be a document available describing any oddities of your system that you should read before writing a serious script.

Appendix E

Isis on the Cheops Processor

This document explains how to use Isis and the structured audio and video and user interface packages on Cheops, and all of the oddities of how Cheops handles these packages.

E.1 Running Isis on Cheops

To run Isis on Cheops, you must first have your environment set up correctly to run Cheops programs. Typing **add cheops** at your shell prompt will accomplish this. Then it is just a matter of running the correct version of Isis using **chex**. There are two versions of Isis available on Cheops: **cheops-isis** and **cheops-isis-iav**. **cheops-isis** does not include any of the structured audio and video or user interface packages—it is simply is a core interpreter for the language. **cheops-isis-iav** is the version that you will probably want to use most often, as it *does* include the structured audio and video and interface packages.

```
% chex cheops-isis-iav

*** Isis (version 2.0)
*** by Stefan Agamanolis
+++ Structured Video Package
=== Structured Audio Package
--- User Interface Package

->
```

Now that you are running the interpreter, you are ready to use all of the special audio, video, and interface functions. Read the documents *Structured Video with Isis*, *Structured Audio*

with Isis, and *User Interfaces with Isis* for information about these functions. The operation of these packages on the Cheops platform is the same as described in those documents, with a few exceptions. These exceptions and other differences are explained in the following pages.

E.2 Structured Video

The following is a list of the oddities of the structured video package running on Cheops:

- The functions `stop-video-system`, `reset-video-system`, and `terminate-video-system` have no effect.
- URL support in `new-video-object` is not present.
- The object class name is ignored in `new-video-object`.
- Only two-dimensional dat files with 3 or 4 channels may be loaded as video objects. 3 channels is assumed to be RGB, and 4 channels is RGB with alpha. There may be other subfiles—see the document *The structo-vision file format* for more information.
- The horizontal (X) dimension of all video objects must be a multiple of 16 greater than or equal to 32. Otherwise, M1 memory transfers will not function properly.
- Window brightness control with the `win-brightness` parameter is not supported.
- Variable opaqueness of actors with the `ac-visibility` parameter is not supported. This parameter must be either 0.0 or 1.0.
- Alpha channels are converted to depth buffers when video objects are loaded. Therefore the `ac-use-alpha` and `ac-do-blending` parameters have no effect. Alpha blending support is not present.
- Scales of actors are limited to fractions where the numerator and denominator are integers less than or equal to 7. Values specified for the `ac-scale` parameter will be converted to the nearest available scale.

E.3 Structured Audio

The Cheops structured audio package requires an audio server to be running on the host Alpha workstation. You can run the audio server with the command `super-alpha-audio-server` which should be in your path if Cheops is in your environment.

If this server is not running when you call `initialize-audio-system` in Isis, you will be asked to run it. The audio server must be killed after your session with Isis on cheops by calling `terminate-audio-system` in Isis or by killing the process manually in your command shell on the host computer.

The following is a list of the oddities of the structured audio package running on Cheops:

- The functions `stop-audio-system` and `reset-audio-system` have no effect.
- The audio server is capable of handling only two-dimensional audio spaces. Therefore, the Y (second) coordinate in all positions and dimensions is ignored.
- Only one-channel raw 44.1 KHz sound files can be loaded by `new-audio-object`.
- You must create the speakers, set their positions, and assign them to an engine alone and call `realize-audio` once before adding other entities to that engine and before calling `start-audio-system`. Also, only two speakers are supported. The first speaker you create will be rendered on the output channel 1 (L), and the second on output channel 2 (R). Here is an example of what the beginning of a script should look like:

```
(load "structured-av.isis")

(set audeng (new-audio-engine))
(set speaker1 (new-speaker))
(set speaker2 (new-speaker))

(aud-update speaker1
   sp-position (Pos -1.0 0.0 -1.0))
(aud-update speaker2
   sp-position (Pos 1.0 0.0 -1.0))
(aud-update audeng
   aud-speakers (AddrList speaker1 speaker2))
(realize-audio audeng)

...

(start-audio-system)
```

E.4 User Interface

The following is a list of the oddities of the user interface package running on Cheops:

- The user interface package connects to the knob box that is typically attached to Cheops. Each call to `new-input` will acquire the next available knob in order from 0 to 7. Therefore, you must create dummy inputs in order to skip certain knobs in that order.
- You can only create one interface. Calls to `start-interface` and `stop-interface` are irrelevant.
- `String` inputs are not supported.

E.5 Example Scripts

The directory `/mas/garden/cheops/isis` contains useful Isis scripts, such as `structured-av.isis` which is helpful in structured audio and video applications. There are also several example scripts and other useful documents located in this directory.

Appendix F

The *Wallpaper* script

This appendix contains a slightly abbreviated version of the script that executes the *Wallpaper* interactive narrative, along with some comments about how it works. Each piece below appears in the same order as it did in the original script.

F.1 Initializing

The first part of the script initializes the entire system. Because of an oddity of Cheops (see Appendix E), the procedure to initialize the audio package is slightly more complicated. The top-level audio engine and speakers are created at this stage.

```
# ------------------------------------------------------------------------
# The Interactive Wallpaper script

(print newline "Welcome to Wallpaper" newline newline)

(load "structured-av.isis")     # Loads standard type definitions

(initialize-interface)
(initialize-audio-system)
(initialize-video-system)

(set audeng (new-audio-engine))
(set speaker1 (new-speaker))
(set speaker2 (new-speaker))

(aud-update speaker1 sp-position (Pos -1.0 0.0 -1.0))
(aud-update speaker2 sp-position (Pos 1.0 0.0 -1.0))
(aud-update audeng
            aud-output-descriptor "alpha_audio"
            aud-speakers (AddrList speaker1 speaker2))
```

```
(realize-audio audeng)

# Since the audio and video coordinate systems are slightly different,
#   the following two variables must be multiplied and added to a video
#   position to convert it to its corresponding position in the audio
#   environment:

(set sound-pos-trans (Pos -2000.0 0.0 500.0))
(set sound-pos-mult (Pos 0.001 0.001 0.001))

(start-video-system)
(start-audio-system)
```

F.2 Creating the video objects

The following piece of script creates all of the video objects that are used in the entire production. Since Cheops supports the DAT file format used at the MIT Media Lab, some filenames contain some letters and numbers between square brackets [ ]. This information specifies a particular subsampling of the data in that file. This odd file specification style is not part of Isis.

```
# ------------------------------------------------------------------------
# Create video objects

# ----------------------
# The attic

(set room25-images
     (new-video-object "model/scene.rgbz" "m1-room25" "background"))

# The following file creates a list called 'the-views' which contains
#   camera parameters for the 25 pre-rendered views of the attic that are
#   used in the production:

(load "model/room25views.isis")

# ----------------------
# John

(set john1-images
     (new-video-object "masjohn/masjohn1.rgba[d /3]" "john1" "john"))

(set john2-images
     (new-video-object "masjohn/masjohn2.rgba[d /3]" "john2" "john"))

(set john3-images
     (new-video-object "masjohn/masjohn3.rgba[d /3]" "john3" "john"))

(set john4-images
     (new-video-object "masjohn/masjohn4.rgba[d /3]" "john4" "john"))

(set john5-images
     (new-video-object "masjohn/masjohn5.rgba[d /3]" "john5" "john"))
```

```
# ----------------------
# Kathy at the beginning

(set kathy1a-images
     (new-video-object "maskathy/maskathy1.rgba[d 30+80/3]" "kathy1a" "kathy"))

(set kathy2a-images
     (new-video-object "maskathy/maskathy2.rgba[d 30+80/3]" "kathy2a" "kathy"))

(set kathy3a-images
     (new-video-object "maskathy/maskathy3.rgba[d 30+80/3]" "kathy3a" "kathy"))

(set kathy4a-images
     (new-video-object "maskathy/maskathy4.rgba[d 30+80/3]" "kathy4a" "kathy"))

(set kathy5a-images
     (new-video-object "maskathy/maskathy5.rgba[d 30+80/3]" "kathy5a" "kathy"))

# ----------------------
# Kathy saying "I don't know"

(set kathy1b-images
     (new-video-object "maskathy/maskathy1.rgba[d 180+50/3]" "kathy1b" "kathy"))

(set kathy2b-images
     (new-video-object "maskathy/maskathy2.rgba[d 180+50/3]" "kathy2b" "kathy"))

(set kathy3b-images
     (new-video-object "maskathy/maskathy3.rgba[d 180+50/3]" "kathy3b" "kathy"))

(set kathy4b-images
     (new-video-object "maskathy/maskathy4.rgba[d 180+50/3]" "kathy4b" "kathy"))

(set kathy5b-images
     (new-video-object "maskathy/maskathy5.rgba[d 180+50/3]" "kathy5b" "kathy"))

# ----------------------
# Dress form with rag

(set dfrag1-images
     (new-video-object "dfrag/dfrag1.rgba" "dfrag1" "dressform"))

(set dfrag2-images
     (new-video-object "dfrag/dfrag2.rgba" "dfrag2" "dressform"))

(set dfrag3-images
     (new-video-object "dfrag/dfrag3.rgba" "dfrag3" "dressform"))

(set dfrag4-images
     (new-video-object "dfrag/dfrag4.rgba" "dfrag4" "dressform"))

(set dfrag5-images
     (new-video-object "dfrag/dfrag5.rgba" "dfrag5" "dressform"))


# ----------------------
# Dress form without rag

(set dfnrag1-images
     (new-video-object "dfnrag/dfnrag1.rgba" "dfnrag1" "dressform"))

(set dfnrag2-images
     (new-video-object "dfnrag/dfnrag2.rgba" "dfnrag2" "dressform"))
```

```
(set dfnrag3-images
     (new-video-object "dfnrag/dfnrag3.rgba" "dfnrag3" "dressform"))

(set dfnrag4-images
     (new-video-object "dfnrag/dfnrag4.rgba" "dfnrag4" "dressform"))

(set dfnrag5-images
     (new-video-object "dfnrag/dfnrag5.rgba" "dfnrag5" "dressform"))

# -----------------------
# Close-Ups of Kathy typing

(set typingcu1-images
     (new-video-object "typing/typingcu1.rgb.10[d 200:240/3]" "typingcu1" "close-up"))

(set typinghands-images
     (new-video-object "typing/typinghands.rgba[d 60:100/3]" "typinghands" "close-up"))

# -----------------------
# Close-Ups of Kathy looking nervous

(set mcukathy-images
     (new-video-object "mcukathy/mcukathy3.rgba[d 20:80/3]" "mcukathy" "close-up"))

(set cukathy-images
     (new-video-object "cukathy/cukathy3.rgba[d 270:430/3]" "cukathy" "close-up"))

# -----------------------
# Close-Up of John grabbing hankerchief

(set hankygrab-images
     (new-video-object "hankygrab/hankygrab2.rgba[d 40:159/3]" "hankygrab" "close-up"))

# -----------------------
# Close-Up of John grabbing rag

(set raggrab-images
     (new-video-object "raggrab/raggrab2.rgba[d 10:129/3]" "raggrab' "close-up"))

(set mcudressform-images
     (new-video-object "raggrab/mcudressform.rgba" "mcudressform" "close-up"))

# -----------------------
# Close-Up of Kathryn with teacup

(set teacup-images
     (new-video-object "cukathryn/cukathryn1.rgba[d 180:220/3]" "teacup" "close-up"))

# -----------------------
# Close-Ups of Kathryn

(set kathryn1-images
     (new-video-object "cukathryn/cukathryn2.rgba[d 20:58/3]" "kathryn1" "close-up"))

(set kathryn2-images
     (new-video-object "cukathryn/cukathryn2.rgba[d 73:133/3]" "kathryn2" "close-up"))

(set kathryn3-images
     (new-video-object "cukathryn/cukathryn3.rgba[d 100:140/3]" "kathryn3" "close-up"))

(set kathryn4-images
     (new-video-object "cukathryn/cukathryn3.rgba[d 265:285/3]" "kathryn4" "close-up"))
```

```
# ------------------------
# Backgrounds

(set clouds1-images
     (new-video-object "backgrounds/clouds1.rgb" "clouds1" "background"))

(set clouds2-images
     (new-video-object "backgrounds/puffs.rgb" "clouds2" "background"))

(set clouds3-images
     (new-video-object "backgrounds/pow2.rgb" "clouds3" "background"))

(set clouds4-images
     (new-video-object "backgrounds/cher.rgb" "clouds4" "background"))

(set kathywall1-images
     (new-video-object "backgrounds/kathywall1.rgb" "kathywall1" "background"))

(set kathywall2-images
     (new-video-object "backgrounds/kathywall2.rgb" "kathywall2" "background"))

(set corner-images
     (new-video-object "backgrounds/corner.rgb" "corner" "background"))


# ------------------------
# The corner graphic used to provide feedback of the current story perspective

(set flick-images
     (new-video-object "flick.rgba" "flick" "overlay"))
```

F.3 Creating the audio objects

This part of the script creates the audio objects, each of which contains a small sound bite (such as a line of speech) or a sound effect.

```
# ------------------------------------------------------------------------
# Create audio objects

(set kathy-1 (new-audio-object "sourd/bites/kathy.1.raw" "kathy-1"))
(set kathy-2 (new-audio-object "sound/bites/kathy.2.raw" "kathy-2"))
(set what-the-heck (new-audio-object "sound/bites/What.the.raw" "what-the-heck"))
(set cant-you (new-audio-object "sound/bites/Cant.you.raw" "cant-you"))
(set what-have (new-audio-object "sound/bites/What.have.big.raw" "what-have"))
(set i-dont-know (new-audio-object "sound/bites/I.dont.raw" "i-dont-know"))
(set i-am-out (new-audio-object "sound/bites/I.am.out.raw" "i-am-out"))
(set look-around (new-audio-object "sound/bites/Look.around.raw" "look-around"))
(set whats-wrong (new-audio-object "sound/bites/Whats.wrong.raw" "whats-wrong"))
(set chimes (new-audio-object "sound/background/newchimes.raw" "chimes"))
(set wind (new-audio-object "sound/background/newwind2.raw" "wind"))
(set typing (new-audio-object "sound/background/newtyping2.raw" "typing"))
(set windandchimes (new-audio-object "sound/background/windandchimes.raw" "windandchimes"))
```

F.4 Creating the structured audio and video entities

This part of the script creates all of the structured audio and video entities that are used in the entire production. Initial values for their parameters are specified if they are different from the default values.

```
# ------------------------------------------------------------------------
# Create all entities

(set videng (new-video-engine))
(set window (new-window))
(set camera (new-camera))

(set videnv (new-video-environment))

(set background (new-actor))        # For the attic background
(set john (new-actor))
(set kathy (new-actor))
(set dressform (new-actor))
(set cubg (new-actor))              # For the background of closeups
(set closeup (new-actor))
(set flick (new-actor))             # The corner graphic

(set audenv (new-audio-environment))

(set listener (new-listener))

(set john-voice (new-voice))
(set kathy-voice (new-voice))
(set bg1-voice (new-voice))         # For ambient background sounds
(set bg2-voice (new-voice))         # For ambient background sounds

(set mainactors (AddrList background john kathy dressform cubg closeup flick))
(set mainsounds (AddrList john-voice kathy-voice bg1-voice bg2-voice))

# ------------------------------------------------------------------------
# Change initial parameter values if necessary

(aud-update audeng
            aud-environment audenv
            aud-voices mainsounds
            aud-listener listener)

(aud-update audenv
            ae-dimensions (Dim 5.0 5.0 5.0))

(aud-update bg1-voice
            vo-position (Pos 0.0 0.0 1.0)
            vo-loop True)

(aud-update bg2-voice
            vo-position (Pos 0.0 0.0 0.0)
            vo-loop True)

(vid-update videng
            vid-output-descriptor "window"
            vid-window window
            vid-camera camera
            vid-actors mainactors
```

```
            vid-environment videnv)

(vid-update window
            win-size (Size 512 512)
            win-location (Loc 320 256))

(vid-update background
            ac-do-zbuffer False
            ac-object room25-images)

(vid-update dressform
            ac-position (Pos 3100.0 0.0 930.0))

(vid-update cubg
            ac-do-zbuffer False)

(vid-update flick
            ac-position-in-3d False
            ac-object flick-images
            ac-depth 100
            ac-location (Loc 480 410))
```

F.5 Creating the user interface

The user interface is created here. Six different inputs are requested and configured, and initial values are stored.

```
# ------------------------------------------------------------------------
# Create the user interface

(set interface (create-interface "Wallpaper interface"))

(set size-input (add-input interface "Size" Int))
(configure-input size-input 128 512 16)
(set-input-value size-input 512)

(set view-input (add-input interface "View" Int))
(configure-input view-input 0 200 1)
(set-input-value view-input 150)

(set cu-input (add-input interface "Closeup" Int))
(configure-input cu-input 0 90 1)
(set-input-value cu-input 90)

(set perspective-input (add-input interface "Perspective" Int))
(configure-input perspective-input 0 90 1)
(set-input-value perspective-input 0)

(set flickvis-input (add-input interface "flickvis" Bool))
(set-input-value flickvis-input True)

(set stop-input (add-input interface "Stop" Bool))
(set-input-value stop-input False)

(start-interface interface)
```

F.6 Describing the master shot

The experiencer is able to move around the room during the master shot. For different positions in the room, different views of the actors must be used. Timelines are used to express which video objects should be used by the main actors for the various viewer positions in the room (which correspond to particular frames of the pre-rendered attic backgrounds).

```
# ------------------------------------------------------------------------
# Describe playout of master shot

# ------------------
# For each actor, designate which objects will be used
#   when which attic background frames are in view

(set split12 3)    # The split point frame numbers
(set split23 8)
(set split34 13)
(set split45 18)
(set split51 23)

(set john-object-fl (new-timeline john1-images))
(key john-object-fl split12 john2-images)
(key john-object-fl split23 john3-images)
(key john-object-fl split34 john5-images)
(key john-object-fl split45 john4-images)
(key john-object-fl split51 john1-images)

(set kathya-object-fl (new-timeline kathy1a-images))
(key kathya-object-fl split12 kathy2a-images)
(key kathya-object-fl split23 kathy3a-images)
(key kathya-object-fl split34 kathy5a-images)
(key kathya-object-fl split45 kathy4a-images)
(key kathya-object-fl split51 kathy1a-images)

(set kathyb-object-fl (new-timeline kathy1b-images))
(key kathyb-object-fl split12 kathy2b-images)
(key kathyb-object-fl split23 kathy3b-images)
(key kathyb-object-fl split34 kathy5b-images)
(key kathyb-object-fl split45 kathy4b-images)
(key kathyb-object-fl split51 kathy1b-images)

(set dfrag-object-fl (new-timeline dfrag1-images))
(key dfrag-object-fl split12 dfrag2-images)
(key dfrag-object-fl split23 dfrag3-images)
(key dfrag-object-fl split34 dfrag5-images)
(key dfrag-object-fl split45 dfrag4-images)
(key dfrag-object-fl split51 dfrag1-images)

(set dfnrag-object-fl (new-timeline dfnrag1-images))
(key dfnrag-object-fl split12 dfnrag2-images)
(key dfnrag-object-fl split23 dfnrag3-images)
(key dfnrag-object-fl split34 dfnrag5-images)
(key dfnrag-object-fl split45 dfnrag4-images)
(key dfnrag-object-fl split51 dfnrag1-images)
```

Next, timelines are used to express values of John's video object, visibility, frame number, and audio objects over the entire running time. The same is done for Kathy, and then the dressform.

```
# ------------------
# Describe John's master shot playout

(set john-objlist-tl (new-timeline john-object-fl))
(key john-objlist-tl 0 john-object-fl)

(set john-visibility-tl (new-timeline 0.0))
(key john-visibility-tl 0.0 0.0)
(key john-visibility-tl 20.0 1.0)
(key john-visibility-tl 50.0 0.0)

(set john-frame-tl (new-timeline 0))
(key john-frame-tl 20.0 0)
(key john-frame-tl 50.0 299 linear)

(set john-sounds-tl (new-timeline Null))
(key john-sounds-tl 0.0 Null)
(key john-sounds-tl 4.0 kathy-1)
(key john-sounds-tl 12.0 kathy-2)
(key john-sounds-tl 20.0 what-the-heck)
(key john-sounds-tl 22.0 cant-you)
(key john-sounds-tl 26.5 what-have)
(key john-sounds-tl 32.0 i-am-out)
(key john-sounds-tl 38.0 look-around)
(key john-sounds-tl 40.0 whats-wrong)

# The following file creates a list called 'john-positions' that
#   contains a three-dimensional position for John for all his frames.
#   (found by projecting his hotpoint onto the floor of the blue-screen
#   stage using camera calibration information):

(load "masjohn/john-positions.isis")

(set john-offset (Pos 1270.0 0.0 780.0))


# ------------------
# Describe Kathy's master shot playout

(set kathy-objlist-tl (new-timeline kathya-object-fl))
(key kathy-objlist-tl 0 kathya-object-fl)
(key kathy-objlist-tl 27.5 kathyb-object-fl)

(set kathy-frame-tl (new-timeline 0))
(key kathy-frame-tl 0.0 0)
(key kathy-frame-tl 5.0 50 linear)
(key kathy-frame-tl 5.01 0)
(key kathy-frame-tl 10.0 50 linear)
(key kathy-frame-tl 10.01 20)
(key kathy-frame-tl 16.0 79 linear)
(key kathy-frame-tl 16.01 0)
(key kathy-frame-tl 21.0 50 linear)
(key kathy-frame-tl 21.01 0)
(key kathy-frame-tl 26.0 50 linear)
(key kathy-frame-tl 27.5 0)
(key kathy-frame-tl 35.5 49 linear)

(set kathy-sounds-tl (new-timeline Null))
```

```
(key kathy-sounds-tl 0.0 Null)
(key kathy-sounds-tl 30.5 i-dont-know)

# Kathy's positions are finely tuned for each of her five views.  A
#   timeline is used to store the optimal positions and to interpolate
#   between them:

(set kathypos-fl (new-timeline (Pos 0.0 0.0 0.0)))
(key kathypos-fl 0 (Pos 2230.0 1070.0 -570.0))
(key kathypos-fl 5 (Pos 2120.0 1090.0 -800.0) linear)
(key kathypos-fl 10 (Pos 1940.0 1100.0 -700.0) linear)
(key kathypos-fl 15 (Pos 2150.0 1040.0 -600.0) linear)
(key kathypos-fl 20 (Pos 2180.0 1110.0 -680.0) linear)
(key kathypos-fl 25 (Pos 2230.0 1070.0 -570.0) linear)

(set kathysoundpos
     (* (+ (Pos 2750.0 1000.0 -870.0) sound-pos-trans) sound-pos-mult))

(aud-update kathy-voice
            vo-position kathysoundpos)


# ------------------
# Describe dressform master shot playout

(set dressform-objlist-tl (new-timeline dfrag-object-fl))
(key dressform-objlist-tl 0 dfrag-object-fl)
(key dressform-objlist-tl 36.0 dfnrag-object-fl)
```

F.7 Describing the close-up streams

There are five different "streams" each of which has a particular playout of close-ups and backgrounds. These five streams cover the story space between John's and Kathy's subjective point of view. Five timelines are used to express the progression of close-ups in each stream—one each for the close-up object, the close-up priority, the frame number of the close-up to show, a close-up mode (z-buffer or no z-buffer), and a background object. The close-up priority ranges from 0 to 10 and determines how small the display size must be in order to show the close-up (the lower the number the more likely it will be shown).

For brevity, only one of the stream definitions is shown below. There are four others that are not shown but are defined analogously.

```
# ------------------
# Neutral perspective (3rd of 5)

(set closeup3-tl (new-timeline typingcu1-images))  # For the close-up object
(set cupriority3-tl (new-timeline 0))              # For the close-up priority
(set cuframe3-tl (new-timeline 0))                 # For the frame number of the closeup
(set cumode3-tl (new-timeline True))               # For the z-buffer mode of the close-up
(set cubg3-tl (new-timeline corner-images))        # For the background of the close-up
```

```
#---
(key closeup3-tl 0 typingcu1-images)
(key cupriority3-tl 0 4)
(key cuframe3-tl 0 0)
(key cumode3-tl 0 False)
(key cubg3-tl 0 corner-images)

(key cupriority3-tl 3.99 10)
(key cuframe3-tl 3.99 40 linear)
(key cumode3-tl 3.99 True)

#---
(key closeup3-tl 4.0 typinghands-images)
(key cupriority3-tl 4.0 4)
(key cuframe3-tl 4.0 0)
(key cubg3-tl 4.0 corner-images)

(key cupriority3-tl 7.99 10)
(key cuframe3-tl 7.99 40 linear)

#---
(key closeup3-tl 15.0 mcukathy-images)
(key cupriority3-tl 15.0 5)
(key cuframe3-tl 15.0 0)
(key cubg3-tl 15.0 kathywall1-images)

(key cupriority3-tl 18.99 10)
(key cuframe3-tl 18.99 40 linear)

#---
(key closeup3-tl 32.5 cukathy-images)
(key cupriority3-tl 32.5 3)
(key cuframe3-tl 32.5 0)
(key cubg3-tl 32.5 clouds2-images)

(key cupriority3-tl 34.49 10)
(key cuframe3-tl 34.49 20 linear)

#---
(key closeup3-tl 34.5 raggrab-images)
(key cupriority3-tl 34.5 1)
(key cuframe3-tl 34.5 0)
(key cubg3-tl 34.5 clouds3-images)

(key cupriority3-tl 46.49 10)
(key cuframe3-tl 46.49 120 linear)

#---
(key closeup3-tl 46.5 cukathy-images)
(key cupriority3-tl 46.5 1)
(key cuframe3-tl 46.5 20)
(key cubg3-tl 46.5 clouds4-images)

(key cupriority3-tl 51.99 10)
(key cuframe3-tl 51.99 75 linear)

#---
(key closeup3-tl 52.0 mcudressform-images)
(key cupriority3-tl 52.0 7)
(key cuframe3-tl 52.0 0)
(key cubg3-tl 52.0 clouds3-images)

(key cupriority3-tl 55.99 10)
(key cuframe3-tl 55.99 0)
```

```
#---
(key closeup3-tl 56.0 cukathy-images)
(key cupriority3-tl 56.0 1)
(key cuframe3-tl 56.0 70)
(key cubg3-tl 56.0 clouds4-images)

(key cupriority3-tl 59.99 10)
(key cuframe3-tl 59.99 110 linear)

#---
(key closeup3-tl 60.0 mcudressform-images)
(key cupriority3-tl 60.0 7)
(key cuframe3-tl 60.0 0)
(key cubg3-tl 60.0 clouds3-images)

(key cupriority3-tl 63.99 10)
(key cuframe3-tl 63.99 0 linear)

#---
(key closeup3-tl 64.0 cukathy-images)
(key cupriority3-tl 64.0 1)
(key cuframe3-tl 64.0 110)
(key cubg3-tl 64.0 clouds4-images)

(key cupriority3-tl 68.99 10)
(key cuframe3-tl 68.99 160 linear)
```

F.8 Expressing the changes in story perspective

The story perspective value ranges from 0 to 9. Timelines are used to express which "streams" defined above will be used for each point in this range. Then, changes in acoustics and ambient sound objects are also specified over this range in a similar manner.

```
# ------------------
# Describe the changes in perspective from John to Kathy

# First use a timeline to express which of the streams will be used
#   for each perspective value from 0 to 9:

(set closeup-pl (new-timeline closeup1-tl))
(set cupriority-pl (new-timeline cupriority1-tl))
(set cuframe-pl (new-timeline cuframe1-tl))
(set cumode-pl (new-timeline cumode1-tl))
(set cubg-pl (new-timeline cubg1-tl))

(key closeup-pl 0 closeup1-tl)
(key cupriority-pl 0 cupriority1-tl)
(key cuframe-pl 0 cuframe1-tl)
(key cumode-pl 0 cumode1-tl)
(key cubg-pl 0 cubg1-tl)

(key closeup-pl 2 closeup2-tl)
(key cupriority-pl 2 cupriority2-tl)
(key cuframe-pl 2 cuframe2-tl)
```

```
(key cumode-pl 2 cumode2-tl)
(key cubg-pl 2 cubg2-tl)

(key closeup-pl 4 closeup3-tl)
(key cupriority-pl 4 cupriority3-tl)
(key cuframe-pl 4 cuframe3-tl)
(key cumode-pl 4 cumode3-tl)
(key cubg-pl 4 cubg3-tl)

(key closeup-pl 6 closeup4-tl)
(key cupriority-pl 6 cupriority4-tl)
(key cuframe-pl 6 cuframe4-tl)
(key cumode-pl 6 cumode4-tl)
(key cubg-pl 6 cubg4-tl)

(key closeup-pl 8 closeup5-tl)
(key cupriority-pl 8 cupriority5-tl)
(key cuframe-pl 8 cuframe5-tl)
(key cumode-pl 8 cumode5-tl)
(key cubg-pl 8 cubg5-tl)


# Now describe how the audio will change from perspective 0 to 9:

(set reverb-pl (new-timeline 0.0))
(key reverb-pl 0 0.0)
(key reverb-pl 9 0.7 linear)

(set sizescale-pl (new-timeline 1.0))
(key sizescale-pl 0 1.0)
(key sizescale-pl 9 10.0 linear)

(set distscale-pl (new-timeline 1.0))
(key distscale-pl 0 1.0)
(key distscale-pl 9 10.0 linear)

(set volscale-pl (new-timeline 1.0))
(key volscale-pl 0 0.0)
(key volscale-pl 9 10.0 linear)


# The ambient sound objects and volumes change over story perspective as well:

(set bg1obj0-tl (new-timeline Null))
(key bg1obj0-tl 0.0 typing)
(key bg1obj0-tl 26.0 Null)

(set bg1obj5-tl (new-timeline Null))

(set bg1obj-pl (new-timeline bg1obj0-tl))
(key bg1obj-pl 0 bg1obj0-tl)
(key bg1obj-pl 5 bg1obj5-tl)

(set bg1vol-pl (new-timeline 1.0))
(key bg1vol-pl 0 -10.0)

(set bg2obj-pl (new-timeline wind))
(key bg2obj-pl 0 wind)
(key bg2obj-pl 5 windandchimes)

(set bg2vol-pl (new-timeline 1.0))
(key bg2vol-pl 0 -5.0)
(key bg2vol-pl 9 5.0 linear)
```

F.9 Creating a help function

This part of the script defines a function called `help` that prints a message describing how to start the presentation.

```
# ------------------------------------------------------------------------
# Help function:

(set help
 (proc ()
  (begin
    (print newline
           "To run the wallpaper movie, type" newline newline
           "      (wallpaper <runmode> <sizemode>)" newline newline
           "where <runmode> is:" newline
           "  play   (for normal play mode)" newline
           "  loop   (for looping play mode)" newline
           "  jog    (for jog mode)" newline newline
           "and <sizemode> is:" newline
           "  static        (for constant size display)" newline
           "  resize        (for resizable display)" newline
           "  intelresize   (for intelligently resizable display" newline newline
           "Knob assignments:" newline newline
           "  -------" newline
           "  | 6 7 |" newline
           "  | 4 5 |" newline
           "  | 2 3 |" newline
           "  | 0 1 |" newline
           "  -------" newline newline
           "  Knob 0 is DISPLAY SIZE (if in a resizable mode)" newline
           "  Knob 1 is SPATIAL VIEWPOINT" newline
           "  Knob 2 is CLOSEUPIVITY (in non-intellignet resize modes)" newline
           "  Knob 3 is STORY PERSPECTIVE" newline
           "  Knob 4 is CORNER GRAPHIC ON/OFF SWITCH" newline
           "  Knob 5 is unused" newline
           "  Knob 6 is PLAYBACK JOG (if in jog mode)" newline
           "  Knob 7 is QUIT TO ISIS SHELL" newline newline
           "*** IMPORTANT:  When you are finished, type (exit) to quit." newline newline
           "Type (help) to see this message again later." newline newline))))
```

F.10 Defining the delivery procedure

This part of the script contains a large function that actually generates the output for the entire presentation using all the elements defined previously. The structure of the function is actually quite simple. It starts by storing initial values for variables and putting the correct actors and voices "on stage." The middle part of the function is simply a loop that begins by checking the current timer and user interface values and then builds a frame of the presentation based on those values. The loop continues as long as the run time has not been exceeded (if not running in loop mode) and as long as the experiencer does not activate the *stop* input.

Parameters inside various entities are updated to the values specified in the timelines and lists defined previously in the script. Since updating a sound object causes the sound to start playing from its beginning, the function keeps track of what sounds are currently playing on each voice so that the voice object will only be updated when it changes.

```
# ------------------------------------------------------------------------
# Procedure to run the whole show:

(set loop 0)
(set play 1)
(set jog 2)
(set allframe 3)

(set static 0)
(set resize 1)
(set intelresize 2)

(set wallpaper
 (proc (runmode sizemode)
   (begin
    (vid-update videng                    # Put our actors and voices on stage
                vid-actors mainactors)    #
    (aud-update audeng                    #
                aud-voices mainsounds)    #
    (vid-update window                    # Reset window size
                win-size (Size 512 512))  #
    (set-input-value stop-input False)    # Reset user interface
    (update-interface interface)          #
    (set stop 0)                          #
    (set-video-time 0.0)                  # Reset internal timer
    (set john-lastsound -1.0)             # Keeps track of current audio object
    (set kathy-lastsound -1.0)            #
    (set bg1-lastsound Null)              #
    (set bg2-lastsound Null)              #
    (set last-persp -1)                   # Keeps track of current story perspective
    (set last-bg -1)                      # Keeps track of current background frame

    (while (not stop)                     # Beginning of playback loop
           (begin

             # Grab current user interface and timer values

             (update-interface interface)

             (set bgframe (/ (check view-input) 10))
             (set cupriority (/ (check cu-input) 10))
             (set persp (/ (check perspective-input) 10))
             (set stop (check stop-input))
             (if (!= runmode allframe)
                 (if (= runmode jog)
                     (set time (check time-input))
                     (set time (get-video-time)))
                 Null)

             # persp-changed flags whether or not the story
             #   perspective has changed from its previous value.
             #   Similarly, bg-changed flags whether the attic background
             #   (the viewer's spatial position) has changed

             (set persp-changed (!= last-persp persp))
             (set last-persp persp)
```

```
(set bg-changed (!= last-bg bgframe))
(set last-bg bgframe)

# Update the display

(if (!= sizemode static)
    (begin
      (set size (check size-input))
      (vid-update window
                  win-size (Size size size))
      (if (= sizemode intelresize)
          (set cupriority (- 9 (* 0.0234375 (- size 128))))
          Null))
    Null)

# cushow flags whether or not a closeup will be shown.
# mainshow is the opposite of cushow:

(set cushow (cast Int
                  (>= cupriority
                      ((cupriority-pl persp) time))))
(set mainshow (- 1 cushow))

# All objects were subsampled temporally by 3 when created,
#   hence the factors of 3 below and elsewhere.

(set johnframe (/ (john-frame-tl time) 3))
(set johnpos (+ (john-positions (* johnframe 3)) john-offset))
(set johnsoundpos (* (+ johnpos sound-pos-trans) sound-pos-mult))

# Update the video objects

(if bg-changed
    (begin
      (set curview (the-views bgframe))
      (set viewrefpoint (curview 0))
      (vid-update camera
                  cam-ref-point (curview 0)
                  cam-normal (curview 1)
                  cam-up (curview 2)
                  cam-eye-distance (curview 3)
                  cam-viewport (curview 4))
      (set viewsoundpos (* (+ viewrefpoint sound-pos-trans)
                           sound-pos-mult))
      (aud-update listener
                  li-position viewsoundpos
                  li-normal (curview 1)))
    Null)

(vid-update background
            ac-visibility mainshow
            ac-frame bgframe
            ac-position viewrefpoint)

(vid-update john
            ac-visibility (cast Int
                                (and mainshow
                                     (john-visibility-tl time)))
            ac-object ((john-objlist-tl time) bgframe)
            ac-frame johnframe
            ac-position johnpos)

(vid-update kathy
            ac-visibility mainshow
```

```
            ac-object ((kathy-objlist-tl time) bgframe)
            ac-frame (/ (kathy-frame-tl time) 3)
            ac-position (kathypos-fl bgframe))

(vid-update dressform
            ac-visibility mainshow
            ac-object ((dressform-objlist-tl time) bgframe))

(vid-update cubg
            ac-visibility cushow
            ac-object ((cubg-pl persp) time)
            ac-position viewrefpoint)

(vid-update closeup
            ac-visibility cushow
            ac-object ((closeup-pl persp) time)
            ac-frame (/ ((cuframe-pl persp) time) 3)
            ac-do-zbuffer ((cumode-pl persp) time)
            ac-position viewrefpoint)

(vid-update flick
            ac-frame persp
            ac-visibility (cast Real (check flickvis-input)))

# Update the sound objects

(aud-update john-voice
            vo-position johnsoundpos)

(if persp-changed
    (begin
      (aud-update audenv
                  ae-reverb-scale (reverb-pl persp)
                  ae-dist-scale (distscale-pl persp)
                  ae-size-scale (sizescale-pl persp))
      (aud-update john-voice
                  vo-volume (volscale-pl persp))
      (aud-update kathy-voice
                  vo-volume (volscale-pl persp))
      (aud-update bg1-voice
                  vo-volume (bg1vol-pl persp))
      (aud-update bg2-voice
                  vo-volume (bg2vol-pl persp)))
    Null)

# Update ambient sound 1 object if necessary

(set bg1obj ((bg1obj-pl persp) time))
(if (!= bg1-lastsound bg1obj)
    (begin
      (set bg1-lastsound bg1obj)
      (if (= bg1obj Null)
          (aud-update bg1-voice
                      vo-end-time -1.0)
          (aud-update bg1-voice
                      vo-end-time 100000.0
                      vo-object bg1obj)))
    Null)

# Update ambient sound 2 object if necessary

(set bg2obj (bg2obj-pl persp))
(if (!= bg2-lastsound bg2obj)
    (begin
```

```
            (set bg2-lastsound bg2obj)
            (if (= bg2obj Null)
                (aud-update bg2-voice
                            vo-end-time -1.0)
                (aud-update bg2-voice
                            vo-end-time 100000.0
                            vo-object bg2obj)))
        Null)

    # Update John's audio object if necessary

    (set john-sound-start (key-prev john-sounds-tl time))
    (if (!= john-lastsound john-sound-start)
        (begin
          (set john-lastsound john-sound-start)
          (set john-soundplaying (john-sounds-tl time))
          (if (= john-soundplaying Null)
              (aud-update john-voice
                          vo-end-time -1.0)
              (aud-update john-voice
                          vo-end-time 100000.0
                          vo-object (john-sounds-tl time))))
        True)

    # Update Kathy's audio object if necessary

    (set kathy-sound-start (key-prev kathy-sounds-tl time))
    (if (!= kathy-lastsound kathy-sound-start)
        (begin
          (set kathy-lastsound kathy-sound-start)
          (set kathy-soundplaying (kathy-sounds-tl time))
          (if (= kathy-soundplaying Null)
              (aud-update kathy-voice
                          vo-end-time -1.0)
              (aud-update kathy-voice
                          vo-end-time 100000.0
                          vo-object (kathy-sounds-tl time))))
        True)

    # Output the frame

    (realize-audio audeng)
    (realize-video videng)

    # Loop at the end of playout if applicable

    (if (= runmode loop)
        (if (>= time 75.0) (set-video-time 0.0) Null)
        (begin
          (if (= runmode allframe)
              (set time (+ time 0.3))
              Null)
          (if (>= time 75.0) (set stop True) Null)))))     # End of playback loop

# If playout ends or the viewer activates the 'stop' input,
#   stop the ambient sounds and exit.

(aud-update bg1-voice
            vo-start-time -1.0
            vo-end-time -1.0)
(aud-update bg2-voice
            vo-start-time -1.0
            vo-end-time -1.0)
(realize-audio audeng))))
```

F.11 Giving control to the experiencer

At the end of the script, once all the playback data and functions have been created, the help message is printed and control is turned over to the experiencer who must follow the instructions in the message to begin playback. When she types `(exit)` the structured audio and video systems are terminated and Isis exits!

```
# ------------------------------------------------------------------------
# Print help message and give control to the user

(help)

(interactive)

(terminate-audio-system)
(terminate-video-system)
```

Bibliography

[App95] Apple Computer, Inc., Quicktime World Wide Web site, http://quicktime.apple.com, December 1995.

[AW94] E. H. Adelson and J. Y. A. Wang. *Representing Moving Images with Layers.* IEEE Transactions on Image Processing 3, 5 (September 1994), 625-638.

[BB95] John Bates and Jean Bacon. *Supporting Interactive Presentation for Distributed Multimedia Applications.* Multimedia Tools and Applications 1, 1 (March 1995), 47-78.

[BeB95] Shawn Becker and V. M. Bove, Jr. *Semiautomatic 3-D model extraction from uncalibrated 2-D camera views.* SPIE Symposium on Electronic Imaging: Science & Technology, February 1995.

[BGW94] V. M. Bove, Jr., B. D. Granger, and J. A. Watlington. *Real-Time Decoding and Display of Structured Video.* Proc. IEEE ICMCS, May 1994, 456-462.

[Bov89] V. M. Bove, Jr. *Synthetic Movies Derived from Multi-Dimensional Image Sensors.* PhD Dissertation, Massachusetts Institute of Technology, June 1989.

[BW95] V. M. Bove, Jr. and J. A. Watlington. *Cheops: A Reconfigurable Data-Flow System for Video Processing.* IEEE Transactions on Circuits and Systems for Video Technology 5, 2 (April 1995), 140-149.

[Cha95] Teresa Chang. *Real-Time Decoding and Display of Layered Structured Video.* M.Eng. Thesis, Massachusetts Institute of Technology, June 1995.

[Dav94] Glorianna Davenport. *Seeking Dynamic Adaptive Story Environments.* IEEE Multimedia 1, 3 (Fall 1994), 9-13.

[Dru94] S. M. Drucker. *Intelligent Camera Control for Graphical Environments.* PhD Dissertation, Massachusetts Institute of Technology, June 1994.

[Eva96] Kathleen Lee Evanco. *Customized Data Visualization Using Structured Video.* M.S. Thesis, Massachusetts Institute of Technology, February 1996.

[Fuh94] Borko Fuhrt. *Multimedia Systems: An Overview.* IEEE Multimedia 1, 1 (Spring 1994), 47-59.

[FVFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Second edition, Addison-Wesley, Reading, Massachusetts, 1990.

[Gra95] Brett Dawson Granger. *Real-Time Structured Video Decoding and Display.* M.S. Thesis, Massachusetts Institute of Technology, February 1995.

[Hig87] Scott Clark Higgins. *The Moviemaker's Workspace: Towards a 3D Environment for Pre-Visualization.* M.S. Thesis, Massachusetts Institute of Technology, September 1994.

[Hor86] B. K. Horn. *Robot Vision.* MIT Press/McGraw Hill, Cambridge, Massachusetts, 1986.

[HRD94] I. Herman, G. J. Reynolds, and J. Davy. *MADE: A Multimedia Application Development Environment.* Proc. IEEE ICMCS, May 1994, 184-193.

[IMC95] Interactive Media Corporation, World Wide Web site, http://www.imcinfo.com, December 1995.

[Ing95] Araz Vartan Inguilizian. *Synchronized Structured Sound: Real-Time Three-Dimensional Audio Rendering.* M.S. Thesis, Massachusetts Institute of Technology, September 1995.

[Jav95] Sun Microsystems, *Java: Programming for the Internet*, World Wide Web site, http://java.sun.com, December 1995.

[Kal95] Kaleida Labs, Inc., World Wide Web site, http://www.kaleida.com, December 1995.

[KHY94] F. Kazui, M. Hayashi, and Y. Yamanouchi. *A Virtual Studio System for TV Program Production.* SMPTE Journal 103, 6 (June 1994), 386-390.

[Mac95] Macromedia, Inc., World Wide Web site, http://www.macromedia.com, December 1995.

[McL91] P. C. McLean. *Structured Video Coding.* M.S. Thesis, Massachusetts Institute of Technology, June 1991.

[ME95] Thomas Meyer-Boudnik and Wolfgang Effelsberg. *MHEG Explained.* IEEE Multimedia 2, 1 (Spring 1995), 26-38.

[MM94] S. R. L. Meira and A. E. L. Moura. *A scripting language for multimedia presentations.* Proc. IEEE ICMCS, May 1994, 484-489.

[Ora95] Oracle Corporation, World Wide Web site, http://www.oracle.com, December 1995.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[PM95] Paul Mace Software, Inc., World Wide Web site, http://www.pmace.com, December 1995.

[Pyt95] Python programming language, World Wide Web site, http://www.python.org, December 1995.

[Rey82] C. W. Reynolds. *Computer Animation with Scripts and Actors.* ACM Computer Graphics 16, 3 (July 1982), 289-296.

[S95] P. A. Subrahmanyam, K. J. Singh, Guy Story, and William Schell. *Quality Assurance in Scripting.* IEEE Multimedia 2, 2 (Summer 1995), 50-59.

[She92] Irene J. Shen. *Real-Time Resource Management for Cheops: A Configurable, Multi-Tasking Image Processing System.* M.S. Thesis, Massachusetts Institute of Technology, September 1992.

[Syb95] Sybase, Inc., World Wide Web site, http://www.sybase.com, December 1995.

[Tam96] David Tamés. *Some Assembly Required: Cinematic Knowledge-Based Reconstruction of Structured Video Sequences.* M.S. Thesis, Massachusetts Institute of Technology, January 1996.

[Wat89] John A. Watlington. *Synthetic Movies.* M.S. Thesis, Massachusetts Institute of Technology, September 1989.

[WR94] Thomas Wahl and Kurt Rothermel. *Representing Time in Multimedia Systems.* Proc. IEEE ICMCS, May 1994, 538-543.

[Z95] Thomas G. Zimmerman, Joshua R. Smith, Joseph A. Paradiso, David Allport, and Neil Gershenfeld. *Applying Electric Field Sensing to Human-Computer Interfaces.* Proc. ACM CHI 1995.