Final Thesis

DreamLife 06.01

presented by

Mark Anderson

David Johnson

Marilenis Olivera

Diane Tanabe

June 2001

California State University, Hayward

Table of Contents

Broad Overview

We want to create a multimedia installation that explores new ways of interacting with and experiencing digital media, both individually and as a group.

In general, most of the installations we have experienced focus their attention on technology rather than aesthetics. Utilizing the core concept of dreams, our overall goal is to create an environment that not only utilizes technology in innovative ways but contains art that is compelling and engaging as well.

Although technology will play an important role in our installation, it will be unobtrusive rather than invasive. Interaction with environment will be intuitive and natural.

Our research has shown that most multimedia installations are essentially single-user experiences. We want to provide an experience that encourages people to interact not only with the space and technology but with each other as well. Our intent is to create an experience that enables participants to understand that we are all connected to each other. That one person's actions can and do affect another's experience.

We want to push the boundaries of traditional methods for interacting with digital media. For example, the Memarena project was interesting in that it was a multi-user experience; however, the human-computer interaction was still based on conventional means.

We also want to use this installation to tell a story, rather than to present disconnected imagery. Our story will reflect user interaction with the environment in subtle ways, so that the story becomes an organic part of the users' interaction with digital media, as well as their interaction with each other. The story will provide a propulsive element, leading the users through the installation, without seeming overly goal-oriented or labored.

In essence, we want to create the thinking person's fun house.

Our Means

Human-computer interaction: As stated before, one of our high-level concepts is to explore both human-computer interaction and human-human interaction, mediated by computer. This is a community project; it is not meant to be experienced in isolation.

Sensors: Sensors allow the entire environment to live. Sensors will be our input method.

Art

Our Subject: Dreams

We have chosen the subject of dreams because it fits with our aesthetic, is intellectually engaging, and relates to our desire to create a group/collective experience for the participants. Initial research into the subject of dreams has determined that although dream interpretation appears to be of a personal nature, the ways in which we experience dreams and the dreams themselves have common characteristics. Distortions in size, time and space and mutations in the laws of nature are often experienced in dreams. In addition to being an artistically fertile subject matter, it will also provide opportunities to explore concepts of time, memory, and cause and effect.

Narrative

Our content is delivered as a story, which the users can affect. Our core concepts include: Cause-and-effect: What one user does affects the space for everybody. The environment reacts to the users interacting with it, within it. Human-computer interaction and human-human computer-mediated interaction: Our users will interact with the digital media via sensor technology. How does this 'invisible' computer interface change a person's interaction with other people?

Specificity vs. Universality

Because the central character's dreams are symbolic images of their waking life, and because this character serves as the first-person through which our users will interact with the environment,the character doesn't serve as the focus point of the piece. Rather, we hope that a transference will occur. By putting the users into a dream-like setting and allowing them to directly affect the dreams that they see, we want to evoke in the users a sense of their own dreams, rather than make a purely descriptive piece about another person.

Sound

Music and Sound Overview

Sound plays a significant role as an active parameter of the user experience in DreamLife. Just as visual immersion is critical to creating a new time/place environment, aural immersion has been sited on numerous occasions[1] as a parameter that presents a double problem: it is even more important than the visual element and is often "neglected as visual's poor step-child.[2]"

The sound and music of DreamLife was created specifically for the installation. DreamLife's aural component is conceived in two parts: content-driven sound and ambient moodscapes, which may or may not include distinguishable content related to the narrative but is designed to create a specific mood or ambient effect. See chapter 3 appendix A for a complete listing of sound files for the installation.

The Concept of Aural Layering
Layering the aural environment with sound built upon sound is made much easier if one layer (voice) acts as a canvas onto which another layer, quite possibly more descriptive in nature and obviously related to the narrative, is projected. An example of how we achieve a layering effect is found in the design of the Sound Station in the external environment. As explained in Chapter 7, the vintage radio has two dials, one tuning and one on/off/volume. We have delegated ambient, non-descriptive soundscapes to the volume knob ö now known as the ambient knob ö and content-related sounds to the tuning knob öÊ still known as the tuning knob.

We feel aural layering is especially important in a multi-user environment because one user might be focused on one sound due to his or her proximity in the space (i.e. internal or external environment) as opposed to another user in another part of the space interacting with a wholly different interface. That scenario obviously demands a multiplicity of environments and therefore multiple aural and visual environments.

Descriptive, Content-driven Sound
As sound and music are highly influential components in creating an immersive user experience, our goal is to compose and integrate music and sound relevant to our narrative theme and its milieu. This takes into account music in the following genres:

1) Hollywood and the silver screen
2) The Golden Days of radio (includes radio dramas and comedies, commercials advertisements, Grand 'Ol Opry, and major radio personalities)
3) Dance Hall/Big Band sound
4) Western-European Classical music, primarily from the 18th and 19th centuries.
5) African-American spirituals, field songs and blues (with that inherent instrumentation)
6) Liturgical music (including wedding and funeral music)
7) Speech (including sermons, parents, friends, etc)
8) Sound effects (from the light humming of lightning bugs to the roar of industrial construction equipment)
9) Indigenous American musical idioms such as Ragtime, Honky-Tonk, stride, dixie-land
10) Latin (salsa) and simple cross-rhythm patterns (3:2) that invoke an image of native south pacific drumming.

Music
Music, defined here as possessing any combination of at least two of the following elements: harmony, melody and rhythm is designed to serve two primary purposes in the installation; 1) To have content-specific features which aid in the user-understanding of the narrative and 2) that which purely atmospheric and/or of an ambient nature and is not at all clearly content specific. Both are intended to enhance the overall user experience.

The music composed for DreamLife is influenced by those genres outlined in section 3.2 above. It is our goal to implement styles reflecting popular musical tastes at key points in the main character's life, i.e. late 1930s, late 1950s and 2000. Due to the personality of DreamLife's main character however, popular music from 2000 is hardly represented because we feel that contemporary styles did not enter our main characters aural consciousness. In other words, most elderly people today will not willingly subject themselves to rap, house, bass and drums, grunge, or most other forms of contemporary popular music. However, some did enter her sub-conscious and so we have subsequently included a quick snippet and various points in the sound environment.

Composition, Modification and Content
The creation of sound for an installation based upon the dream-life of an individual must take into account that character's milieu, socio-economic environment, as well as her personal tastes and personality as a whole. In the creation of sound for this environment, we have striven to capture the personal and environmental characteristics of the narrative's central theme.

Achieving this involved composing and improvising music relevant to all the above-mentioned factors as well as modifying existing or pre-composed music (all within the public domain due to their age). For example, having lived all her life in the Bible belt region of the United States, religion played a rather profound role in our main character's life. In order to portray this profundity, we have chosen

music as diverse as Lutheran Chorales harmonized by Johann Sebastian Bach (1685-1750) and African-American folk songs without any notated sources in existence. In the case of the latter, we have taken melody, harmony and rhythm of these songs and re-recorded them using modern synthesizers and digitized audio, usually in the form of vocals, guitar and violin (fiddle).

Another element in the composition of music and sound for DreamLife was discovered on the Internet.Ê As Carl Jung believed that dreams encapsulated, among other things, archaic remnants of our primitive selves, we believe that cultural artifacts can also be handed down through dreams. This is relevant in a primarily Judeo-Christian sense but can also involve secular information. Of great interest in this realm was a site devoted to Alchemy and, specifically, to a book, first published in 1617 in Latin, by Michael Maier called Atlanta fugiens. As the site explains:

It was a most amazing book as it incorporated 50 emblems with epigrams and a discourse, but extended the concept of an emblem book by incorporating 50 pieces of music the 'fugues' or canons. In this sense it was an early example of multimedia.[3]

Of great interest were the fugues or canons associated with each image and text. In keeping with our afore-mentioned belief in secular archaic remnants being manifest in dreams, we have felt it relevant to include the subjects or main themes from a number of these pieces as a sort of cantus firmus over which we have designed modern dreamscapes and synthesized textures. We feel this is also an appropriate mix of antiquated and modern sub-conscience elements which, when combined, yield expressive musical qualities.

Hardware and Software Tools for Music and Sound Creation
Much of the music and sound for DreamLife was composed originally for the installation. However, a number of American folksongs spirituals, specifically of the southeast United States, and hymns from the Methodist and Baptist churches were used and re-arranged to fit into the overall aesthetic of the installation. For example, the spiritual Tryin to Make Heaven My Home was re-created with original vocals (the author) manipulated in Digital Performer to alter register so that the result sounds like an alto female with supporting male voices (in four-part harmony).Ê Under the singing is a drone and simulated hand-clapping generated by a Roland JV-1080 Synthesizer. Table 3.1 below lists software programs used in the creation of aural media for DreamLife. Table 3.2 lists hardware and instruments used in DreamLife.

All of the spoken dialogue in the installation was acquired using JVC XM-R70 MiniDisc recorder and an AudioTechnica AT822 stereo microphone as listed below in Table 3.2. Some ambient sound effects such as traffic, school playground, and nature sounds were captured using this same equipment. This equipment is extremely lightweight and portable and is therefore an especially convenient and effective system for most field recording situations.

Table 3.1 - Audio Production Software used for DreamLife

The Installation Sound System and Design
The physical sound design consists of the following hardware (in order of signal flow):

1) Computers and their respective soundcards with stereo output (minijack)

2) a Mackie 1604 16 Channel mixing board

3) [6] Genelec Self-powered mid-range speakers. 5 mid-ranges plus one sub-woofer.

Although the system is quite simple, using the aux outputs to control the amount of signal to each individual speaker. This is extremely necessary in the case of the "radio/sound station." In our early testing of the installation, users at the radio complained that they were not aware of the effects of their actions on the space because they could not hear the radio distinctly. At that time the audio for the "external" space produced via interaction with the radio was routed to a stereo pair of speakers positioned on the periphery of the space. In other words, the sound was dispersed and integrated into the "movie soundtracks" or internal sound to such an extent that the blend prevented the sound station user from gaining clarity and understanding of his or her actions.

At that time, we positioned a speaker under the table that the radio sat on. Also at that time (mid-late May, 2001), we still were directing the left/right stereo outputs from the soundcards into channels 1/2 (L/R) for the "canopy/internal sound" machine and channels 3/4 (L/R) for the "radio/external sound station" machine. Whatever was panned into the left side (channel 3) from the sound station inputs was sent to the speaker under the table and this gave the sound station user a much better sense of control and feeling of results pertaining to his or her actions while manipulating the radio. However, too much valuable sound was being delegated to a speaker under a table...covered by a table cloth. Hardly an optimal setup.

Then it was decided, with the help of Tomonori Yamasaki, to by-pass the main outs (external/sound station)and alt 3/4 outs (internal/movie sound) and connect each of our five available self-powered Genelec speakers to an individual auxiliary output on the Mackie mixing board, which has six "aux" outputs. Using this method, we created a matrix of sound outputs as outlined in figure 3.3.

Figure 3.3 Various aspects of the audio component for DreamLife. The green area represents the physical installation and speaker placement; the yellow and blue areas the aux and mains output matrix; and the red the general signal flow through the Mackie 1604

The Radio Speaker and Sub-woofer
In preliminary testing of the installation, we found that sound station users needed a localized sound source to identify their interaction with the radio. As the original radio speaker was rendered useless with the fitting of sensors to radio, we needed a way to imitate localized audio for the sound station user. Without this, the users felt as though their efforts had no effect on the total environment, or worse, the radio itself. Therefore, we positioned a Genelec speaker directly under the sound station table. The Radio Speaker was connected to main out L and sub-woofer was connected to main out R . That way we could maintain direct control over levels requiring specific control outside of the more generalized room audio system (aux outputs 1 - 4).

Evaluation
My evaluation is based upon my subjective judgement of whether or not my personal, artistic and technological goals for my areas of responsibility for DreamLife were fulfilled. As these areas were wildly varied not only from one another but even within themselves, I feel that the challenges were certainly real and intense.

As my background is in music and as a professional musician, I feel that to judge myself against a yard stick with which I am all too familiar might create conflicts. While I do not consider myself a composer in a serious and disciplined sense, I do tremendously enjoy the marriage of video and audio and the rewards that come with this success.

In totally, I feel that, as a single, homogenous piece, DreamLife had moments of tremendously successful unions between the aural and visual components. The vast majority of these moments occurred within the QuickTime movies in which audio and video had been sculpted as one. However, between the internal and external environments, due to the interactive nature of the piece, these unions were purely temporal, ephemeral and, at best, temporary chance occurrences of beauty...or ugliness.

As I expressed to James Petrillo on more than one occasion, in the world of the real installation, sound is given solely to not only one but often a team of individuals. If manpower is proportionate to sound's importance, and I believe it is, this approach is totally justified - if ideal. In addition to deploying sensors, designing and implementing electronic circuits, and shooting, editing and manipulating digital video, I was ultimately responsible for how this installation sounded. To dramatize the point, taking this all into account, I am happy that we had any satisfactory sound at all!

As for my own work in DreamLife, I suppose it would be most revealing to ask my colleagues and the visitors who experienced it what they candidly thought about those areas which were my responsibility. However, I will say that taking into account the breadth of artistic and technical challenges that I faced in helping to create DreamLife, my experience has been ultimately valuable and very fulfilling.

As for the work as a whole, I truly think that DreamLife 06.01 has beautifully succeeded in fulfilling it's mission: to create new narrative forms in an interactive space. Furthermore, judging user response and feedback, it was clear on June 6, 2001 that DreamLife was an event of intriguing multi-user concepts, successful user-interfaces, and compelling visual and aural media. In fact, the space itself was visually tantalizing thanks to Diane Tanabe's highly artistic vision and her ability to realize that vision. In addition, Marilenis Olivera's contributions on artistic and technological fronts enhanced every aspect of DreamLife.

Furthermore, the corner-stone upon which DreamLife was built, and where its success ultimately lies, was in a strong and convincing narrative that made content generation fluid and natural. For these imaginatively literary gifts, for the amazing skills as a Lingo programmer that created a technologically stable and highly interactive installation, and for talents that range from the musical to the visual, I would like to express gratitude to David Johnson.

With the combined talents of these people, DreamLife was destined to succeed.

Video
Overview
The major component of our art work is a series of QuickTime movies that we created to be shown in the space. These movies are displayed both on the Bed canopy and against a side wall. As described in the chapter on software architecture, Macromedia Director provides the interactive environment that determines which movies are displayed.

Each page of the Book details a period of Emily's life. We created movies for each period, so that turning a page in the book causes the movies associated with that period to be displayed.

Our Tools
Our work was shot on a pair of Sony mini-DV cameras: a TRV-900, and a VX-1000. We used Adobe Premiere and Apple Final Cut Pro

for editing, and Adobe After Effects for compositing and effects. In addition, we created additional audio tracks for each movie, since one aspect of the user-interface is that a participant can change from the main audio track to an alternate audio track by applying pressure to a bed sensor. We used SoundEdit 16 and QuickTime Pro to add the additional audio track to each movie.

In order to ensure that the movies could play off the hard drive without stalling, we used Media Cleaner Pro to compress them. We specified bit rates ranging from 4Mbps to about 8Mbps, depending on the source material, although Media Cleaner frequently used less than the specified rate (for example, "Father" was encoded at less than 1Mbps average).

Finally, we used Softimage 3D to create the 3D models for "The Lightning Bugs," and Terragen to generate the landscape scenes in that movie.

Our Movies
These are the movies we created.

Mother_funeral
Emily's dream of her mother's funeral begins almost as a social-realist documentary, recalling such movies as "The Grapes of Wrath" and "You Only Live Once." It soon turns into a nightmare in which she's overcome by the voices of the "graveyard angels," the statues that fill the cemetery. Her father keeps calling her back, but he finally loses her as her mind drifts away.

Lightning_bugs
This childhood dream of loss and homelessness starts with Emily's last waking memory of her father saying goodnight, and then she envisions a swarm of lightning bugs, desperately attempting to talk to her. As they drift away, she hears them say "We have no home" and finds herself echoing the words.

Moviestar
Emily fantasizes that her mother is a famous moviestar who lives in Hollywood and goes to fancy premieres and parties all the time which is the reason why she is no longer present in Emily's day to day life.

Privilege
The privilege trilogy are meant to describe Emily's relatively comfortable circumstances as a young girl in relation to much of the country's suffering in the late1930s. As a result she participated in piano lessons (something her mother had also excelled in as a child and carried into adulthood).

privilege_1.mov - This movie exhibits not only the sense of pride and privilege Emily felt as a child at having become an accomplished student of piano, but also the trepidation and anxieties that surround her performances in local piano recitals. Most families at this time had to sell their pianos for survival money.

privilege_2.mov - Privilege 2 still uses 19th century romantic piano repertoire to create a mood of comfort, cultural awareness and privileged upbringing. There are also glimpses of the nature that surrounded her and permeated her conscious and sub-conscious lives. At one moment, we see Emily run down a wooded path. As most children, she is capable of rebelling and in this case, ran away for a short time because of a conflict with her mother.

privilege_3.mov -This piece centers on her father as the central figure of her privileged existence; his easy disposition, his love of his new 1938 Ford, the circumstances that enable them to enjoy motoring on a Sunday afternoon. Again, we glimpse her propensity to rebel.

Carrousel
Due to the death of her mother and the fact that before her mother died Emily was arguing with her, Emily experiences a dream where she is accused of the death of her mother. The carousel dream starts as a nice experience and it ends with a series of characters from the carousel that accuse Emily.

Playground
As any child Emily enjoys playing in the playground. The playground dream reflects Emily's feelings of isolation. She is always left alone, in school or in the theatre, by her friends and her brother Jed. In this dream she enjoys the playground but at the same time she feels isolated.

Nightmare
This dream sequences explores the fear associated with feelings of being abandoned. Emily has a nightmare and searches for someone to comfort her but discovers that she is all alone.

Father
This is a brief vignette, showing a single image of Emily's father reading before the fire, listening to "that terrible music he loved."

Wedding
This is a text based dream which deals with the anxiety Emily faces trying to plan her wedding.

Water
While pregnant with her daughter Sarah, Emily suffers from anxiety about whether or not she will be a good mother. This dreams deals with different water imagery since water is a symbol for new life and rebirth.

Lostbaby
While pregnant with her daughter Sarah, Emily suffers from anxiety about whether or not she will be a good mother. This dreams deals with he fears of loosing her baby.

Dreamscape
The dreamscape trilogy is meant to represent Emily's middle period, a time of uneasiness at best and sometimes bordering on depression (although in 1958, "depression" was certainly not a condition that elicited the attention it does today). The movie is punctuated with a re-occurring vision of Emily seeing herself dreaming (lucid dreaming) from above. She sees that the dreamer (herself) is very disturbed by the sudden movements and uneasy jerkiness of her feet and legs and desperately tries to find out "what's wrong....what's the matter" - in vain. This is further meant to represent her outward waking oppression of her concerns (typical of a housewife in the 1950s) by the fact that she does not respond to her own pleas and the comment at the end of the movie about going "to Atlanta this spring." Additionally, she is very aware of her socio-economic circumstances and the events that trigger her solemn mood; the selling and subsequent destruction of the family home and all its memories; her wedding (representing the "beginning of the end"); and the lack of worldly experiences and her fantasizing of far-off, exotic places.(dreamscapes 1,2 and 3 are essentially modified excerpts from dreamscape_all.mov)

dreamscape_1.mov - This movie emphasizes nostalgia embodied in the destruction of her childhood home.

dreamscape_2.mov - This is a quick glimpse into her discontent based upon her provincial view of the world. Being an intelligent woman, she is forced to understand her predicament yet live happily within it. Images of Moai on Easter Island and her overhearing a social anthropologist professor from the local university discuss his recent travels at a dinner party at once fire her imagination cause her torment in her dreams.

dreamscape_3.mov - is a continuation of dreamscape_2.mov

BTA
In the 1950s, Emily finds herself stifled by conformity and envisions a game show host from a typically invasive Fifties game show, "Better Than Average," probing her with both nonsensical and terribly personal questions. She finally is led to say that her marriage was "the morning of the world," but instinctively realizes that it's a lie even as she's rewarded for the answer.

Destruction_of_home
This is a dream that Emily had when her brother Jed sold their family home. In this dream she sees how all her memories are gone with the destruction of her home.

Acceptance
This movie is intended to show Emily's coming to terms with her final phase and passage into death. After a lifetime of brooding unhappiness, she is overcome by a strong sense of inner peace and relinquishing of responsibilities, mainly having to do with memory as she realizes memory carries with it heavy responsibilities.

Accept_short
This is a completely different "short" from the longer "acceptance.mov." Although it uses similar voice-over, it's imagery is different and mood a little more tense, as if our character is not quite ready to accept her impending death.

Hi_grandma
A dream of her walking in a forest and encountering her two granddaughters jogging. The aged Emily proceeds to offer an unsolicited soliloquy on the relevance of the past.

Peace
Emily dreams of her final Thanksgiving dinner with her family, all of whom know that she doesn't have long to live. They wheel her around the graveyard to see the family plot one last time, where she will shortly be buried alongside her husband and parents, but she finds herself fascinated by the graveyard angels again. This time, she doesn't fear them (as in "Mother_funeral"); instead, she pities them for their disrepair, and finds herself unable to achieve peace until she's willing to give up that pity, along with all other temporal concerns. She sees her funeral and a final moment in a type of Heaven, where a child assures her that "once the forgetting starts, you don't miss anything at all."

Stone
Stone.mov is a short piece that takes place at the end of her life. Emily is being wheeled around the cemetery and is overcome by two feelings: on the one hand, the abundance and apparent permanence of stone and on the other, the transformation of stone back to flowers evidence of her ability to see the big picture at the end of her life.

Narrative

Overview
"Our content is delivered as a story, which the users can affect. It conveys a series of dreams, seen from the viewpoint of a woman, Emily, at three major phases of her life: her childhood, her adulthood, and her own age. The users will affect the narrative as they interact with the installation, in essence creating a new dream narrative of their own from the pieces of Emily's dreams. By putting the users into a dream-like setting and allowing them to directly affect the dreams that they see, we want to evoke in the users a sense of their own dreams, rather than make a purely descriptive piece about another person.

"One of the basic rewards of storytelling is the way it propels its audience from the beginning to the end. Because the narrative is branching, we can't rely on sequence to play its traditional role in supplying order and meaning to the story. Because the interface we're employing to tell our story will be novel, we are designing our narrative simply, so that the user isn't kept in the uncomfortable position of not knowing what to do. It should be quickly evident to the user that a story is being told, and that the user has the ability to alter the sequence of the story and the overall mood of the installation.

"We also recognize that many users may experience only fragments of the story, and so we need to craft our artwork so that it can be satisfying in small pieces. While the connection of the artwork to the larger story affords a different, richer level of satisfaction than the artwork alone, dividing the work into dreams allows us to present it in a modular fashion. Whether a user chooses to spend a minute or an hour within the installation, we want to create a distinctive experience, and the simplicity of the narrative will allow us to do this quickly by lessening the time that the user spends trying to decipher the meaning of the work."

The description above, which was originally written for our application to SIGGRAPH, is a condensed version of the plan that we outlined in our initial thesis proposal. It does a reasonably good job of describing our final approach: experiential rather than didactic, evocative rather than descriptive, interactive with regards to sequence but not actual events (that is, the participants can't change the overall plan of Emily's life, but they can change the order in which her story is told, which in turn changes the meaning of her story).

As we stated in our thesis proposal, our primary concern is dreams, with attendant issue of time and memory.

Emily
The genesis of the story for DreamLife was a short story that David wrote as an undergraduate at San Jose State, called "The Lightning Bugs" (see below). The story described a dream that a little girl, Emily, has following the death of her mother, and is set in the South during the 1930s. The team members developed that story into a 3-part life story, carrying Emily's story into her adulthood (during the 1950s) through to her old age and death in the present day.

After we developed the roadmap of Emily's life, we created a journal that Emily kept, with entries from the three periods covered in our installation. This journal is also included below, and served as a guideline for much of our narrative development. Some of the movies we made were direct dramatizations of the journal entries ("BTA," for example); others, such as "Hi, grandma," included lines from the journal.

Finally, though, the major outline of Emily's life is our artwork itself. The QuickTime movies described in the "Video" chapter take off from the narrative as described in the journal, and lead Emily into personal areas far beyond the skeleton of the journal. In addition, choices we made during our shooting (such as the choice of voice-over actresses for Emily) colored Emily's character as well. Therefore, our narrative is more visual, aural, and ultimately cinematic than literary, particularly because the literary component is only perceived by the participants in the installation as brief epigraphs in the book.

The Lightning Bugs
"She just watches the fireplace," Jed said quietly.

Father looked down at her. "She never makes a sound. I don't know what she's thinking anymore."

"She was talking about the lightning bugs again last night, before she went to bed. I think she waits up to see if they'll come." Jed swallowed. "She's been dreamy like this for the last couple of days."

"Your mother used to get the same way." Father walked over to Emily and knelt down. His face was pressed up close to hers. "Emily? Do you want to go to bed?" he whispered.

“Just a little while longer,” she whispered back.

He backed away and nudged Jed. “Bed time,” he murmured.

“I’m older than she is!” Jed protested.

Father paused. He thought for a second and then nodded. “I’ll take you two up together.”

They stood together for a moment. Father hesitated and then crouched down by Emily. They watched the fire together. The room was warm and red. The couch and the curtains glowed. The rug was soft, and Emily rocked slightly, sitting with her legs crossed.

“Honey?”

She shifted slightly.

“Honey? Don’t you want to see the lightning bugs tonight? I’ll take you up to your room.”

She said nothing. He wrapped his big arms around her gently. She fell into them, and he picked her up, pressing her little head into the crook between his shoulder and his neck. He stood up and began to climb the stairs. He was careful to set each food down softly, not wanting to jolt her.

She could hear Jed leading the way up. She rubbed her nose against the cotton shirt collar. She liked the way he smelled. His cheek was smooth. She pressed her head against his and felt herself rise up the stairway like an angel.

They reached the top of the stairs and she shut her eyes tightly. She knew he would turn left first, into her room. She felt him drop her into one arm as he reached to open the door. She thought he’d press the button for the lights and she waited for the snap and for the inside of her eyelids to glow red. Instead, he slid gently into the room, feeling for the bed. He knelt down and cleared the toys off the bed with his free hand. She held on tight as he drew back the sheet, and then he swung her softly down. She held on for a half-second longer and then let herself fall onto the soft mattress.

He pulled the sheet up over her. Once she was safely in place, he brushed her long bangs aside and pressed his lips against her forehead. “Goodnight, little one,” he said huskily, and he stood to go.

“Daddy?”

He waited. She could hear him hold his breath. “Yes, Emily?”

“Will I see the lightning bugs?”

“Of course you will.” She heard him back away to the door, and then he stopped. “Only . . . don’t wait up too late for them. Okay?”

She didn’t answer, and he turned and closed the door behind him. She heard Jed and Father murmur for a moment, and then all was silence.

She opened her eyes and let them adjust to the darkness. She felt so awake. She could look out the window to her right and see the sky and the outline of her magnolia tree. She knew the hills were miles away in the background, but you couldn’t see them on a dark summer night. She yawned and felt the wave roll gently through her. She lifted her eyes from the windowsill to the stars and back again. Another wave broke over her, and then another. “You could imagine the hills into place,” she thought as the last wave crashed.

And there they were! They were clear as mid-day, lit in bursts by a flashing, flickering ball of light hanging over them. She calmly sat up in bed. The flickering ball was growing larger. First she realized that it was coming closer to her. Then she saw it wasn’t a ball but a long trail of flashing lights. It swayed lazily back and forth, like a sun-drunk moccasin on the surface of the creek.

It was the lightning bugs.

Emily floated out of bed and brought her face to the screen. The pond about a mile from the house blazed bright gold beneath the train of lightning bugs. They lit up the pasture, the sturdy magnolia tree, the porch. Finally, thousands upon thousands of them lit down upon the screen. They wedged themselves in tightly, brilliant flashes reflected in the room behind her. She looked directly at them and was amazed to find her eyes taking on all their fire without pain.

“Where did you come from?” she whispered hoarsely.

A clattering of wings answered her, like metal shutters banging against the aluminum sides of a house.

“Where?” she repeated. She felt them surge anxiously, trying to answer her, but the only sound was the metallic clatter. It grew louder and louder, almost beating its way into words.

And then one lightning bug, divided into huge black sections that sucked up the light, seemed to stand in relief against the others. It must have been there all along—in fact, it now seemed to Emily to have always been right before her. Its sudden appearance rolled farther and farther back into the unreal past.

Now Emily listened within the swarm for the one lightning bug. Its voice broke over the tapping of wings. She could hear it talk in a steely, breathy wheeze. It sounded like a file scraping against the thin blades of a threshing machine.

“I can’t understand you,” she pleaded.

The wings erupted again in a clatter. The large bug danced across the screen in dismay. It searched in vain for vacant holes to plant its legs, which twitched nervously.

“We c-c-came here,” it began slowly, “b-b-because we w-w-w-”

Wisps of metallic voice floated against Emily. She fought hard to hear the words and the song. They seemed to make sense; she felt herself anticipate each sound. She knew what the bug was saying because she could feel herself control his words.

“We w-w-wander each n-n-night,” it continued, despairing at the effort.

“I wanted to see you,” she whispered fiercely.

The noise of the wings died down. There was a soft hum, almost a lowing, like the passionate murmur of mournful cows. The chief bug turned in agonized, fretful circles. It paused, attempted to stutter a reply, and broke again into its frenzied dance. The wings started to beat again. Just as Emily felt the sudden fear that they would leave her, she saw a few lightning bugs drop from the screen in a cluster and open their wings in flight.

“Wait!” she cried.

The huge mass fell from the screen in a ball. The ball clung together as it fell, and then broke into thousands of separate lights, which blossomed out and then regathered in a rapidly receding train.

Her chest started to well with an amazing, tangible sadness. She fought for breath and fought to keep it down. The train danced slowly away and the huge twisting sadness burst out of her.

“Please! Don’t go!” she screamed hysterically. Her cheeks were suddenly covered with tears.

The huge bug pulled itself up from below with heavy chuggings of its thin wings. Emily moaned for breath and the bug seemed to sway in nervous compassion. It waited and then spoke slowly, trying one last time to make every word clear to her: “We . . . have . . . no . . . home,” it said and was gone.

“No . . . home,” rose the low, mournful murmur from the train as it slid from side to side. The lights brushed the tree and fell flickering into the lake of fire.

The Journal

The Journal is divided into three parts. One of our tasks was to develop a convincing voice for Emily in each of the three sections. Our concerns included:

The times in which Emily lived: 1930s language for the childhood section, post-war language for the adulthood, etc.;

The region: Emily is a Southern woman;

Her background: Emily comes from a well-to-do family;

Her circumstances: Emily’s mother dies; Emily marries into a comfortable middle-class life at just the time that America was idealizing comfortable middle-class life; her brother inherits the house and sells it to the Army Corps of Engineers, etc.;

But finally, most importantly, her character. Emily is consistently imaginative and intelligent; she’s also unfulfilled and divided, at least until the end of her life.

Finding this changing and unchanging voice was difficult. For example, the breathless writing she does as a child has to evolve convincingly into the clipped, terse statements of her adult years, and then re-emerge in the funny, slightly wild old woman she becomes. Again,

our final result is NOT a literary piece, and so we were aided by casting and visual decisions. If Emily is convincingly Emily throughout the installation, it's as much testament to the talent of our actresses, and our talent as visual artists, as it is to our conscious decisions in crafting the story.

1938

June 2, 1938, Thursday

Mother says that she's going to take us to the picture show this Saturday, and we all know there's going to be a special act in town and Jed is just DYING to see it and we've been waiting and waiting, although I really don't care for the acts that much. What I love are the pictures. I do understand that they're going to finally make a picture of Gone With The Wind—my favorite book in the whole world! And that Mr. Clark Gable will be in it, and he has been my dream of Rhett Butler since when I very first read the book, and now I've read it three times and I'm going to read it again this very summer. They must be making the movie at this very moment. Mr. Gable must have the reins wrapped around his big hands, and he's flying through the smoke and the red, red glare, and Atlanta burns all around him and Miss Scarlett clings to him with a wild look in her eyes. I can make a look just like that.

I just walked up to the mirror and I can see now that I would be a perfect Scarlett O'Hara. Of course I'm too young, but some day soon we could have an announcement, that Emily Rose Harper is pleased to be visited upon the city of Atlanta, and there will be a great hush and I shall appear at the top of the long flight of stairs and everyone will look up to see me. Of course, I shall be modest, but I'm well aware of the stir that I cause as I glide down the stairs and I know that Jed says that announcements are so much nonsense, but honestly what does he know? Boys don't know anything about it. Clearly Atlanta is the great city, and I know that I shall be well received there.

June 4, 1938, Saturday

So it's Saturday, and Mother took us down to the town to see the picture show, and I saw Annabelle and Beth at the theater with some friend of theirs who will be at our school next year. I wanted to talk to them, but they were all talking to each other and whispering and laughing, and I knew they were talking about boys and I didn't want to.

When we went into the theater, I could see that the picture was going to be a Western, and Jed loves Westerns the most out of all movies but I don't very much. I wanted to see one of the new color movies, and so I tried not to be disappointed but Jed asked me why I kept making a face like that, and I didn't talk to him but I kept looking away, and I could hear those girls giggling and whispering and finally Jed said OK, just be by yourself then, and I was mad and a little sad and I just followed him into the theater.

So when the lights went out, they started out with one of the cartoons, and everybody liked that because it was very funny, and then they brought the lights up for the show, and it was one of those stupid comedy shows! I was hoping it was going to be something magical, like one of those ladies getting sawn in half and then she turns into a bird and flies away, but it was all these stupid jokes and men throwing things at each other, and one man was dressed as a lady, and most of the kids laughed but Jed laughed so loud that I became really embarrassed and tried to look away, like he weren't my brother, even though everybody knows that he was.

Finally, the movie came on, and it was OK but it was just a Western. When it was over, a bunch of the kids were going to go down and play some ball at one of the yards but Mother came to pick us up, and I didn't want to go anyway because it was still really hot even though the afternoon was halfway over. She asked us if we liked the show, and Jed talked and talked and I didn't want to talk. And now I can smell supper.

June 7, 1938, Tuesday

There isn't anything to do today! It's too hot to go out, and I just feel like summer will just go on and on, and I almost feel like I wish I was back in school instead of this big ugly house.

I'm going to begin the fifth grade in September, and everybody knows that I'm doing much better than Jed was when he was in my grade. I keep thinking that Jed is going to get mad at me, because Father had a long talk with him last month about how he's doing in school, but Jed says that it doesn't matter how you do in school. He says what matters is that you know how the world is. And I said, how is the world, and he said that the world is different for different people. He said that it's different for me because I'm a girl, and it's different for him, and it's different for coloreds and white folk, and it's different for poor people, but that he's got the best chance of finding out how the world really is because he's not poor and he's not colored and he's not a girl.

June 8, 1938, Wednesday

I don't want to spend any more time in this hot ugly house!! I'm so tired of it, with its big stairs and its big ugly curtains all over and the heat just all over. I want to go to Atlanta, where they have electrics that make the room nice and cool, and a picture show on every corner. I told Mother about Atlanta, and Mother was already cross because of the heat and we've been seeing each other all day long since school was over, and I think she was feeling a little specially mean because she started telling me that Atlanta was hot and crowded and that I was very silly for thinking that they have it so much better. And I said that I'm not silly, and she started saying not to talk so loud, because I was making her mad. And I said that she was making me mad too, and she got really mad and told me to run upstairs to my room and I could spend the morning thinking about the right way to talk to your parents. So I walked up here and stomped on every step on the way

up the stairs, and she yelled something at me but I didn't hear what it was.

I hate it here. I hate listening to the radio, when I want to listen to the NBC radio theater where Bing Crosby sings, and I remember hearing the girls at school say that Mr. Bing Crosby would be a wonderful husband, because he makes fifty thousand a year, and I said that I don't care how much he makes, and I don't care about a husband, because I just want to hear him sing. And that Hetty laughed at me and said that I don't care about a husband because I'm just a baby, and she made a baby face and everybody laughed.

And it doesn't matter that I want to listen to the NBC radio theater, because Father always wants to hear the Opry and Uncle Dave Macon, and I don't want to hear that country music. When you see go to the pictures, and the Countess of Romania is on a boat to lead her people to Freedom, and her special Count walks up behind her on the boat to sing a love song to her to prove to her that he's the only man who can make Romania free, and he's singing with a big orchestra. Uncle Dave Macon always sings those corny old-time songs with a fiddle and I don't want to hear that. And Jed just wants to listen to his mystery shows, and Father lets him since they're on after the Opry.

It's later; I just put down my diary because Mother came up and told me she was sorry about yelling at me, and she wants to go down to the lake with me, and we can take our little boat onto the water. And I said I'm too hot, and anyway who cares about a little boat, and our ugly little lake, with its little lily pads and all the flies? And Mother got cross with me again and said that she's going out onto the lake, and if I want to come then she'll let me, but she's not going to wait for me. Some times I like the lake, but I just want to be away from here right now. And I can hear Mother leave out the back and tell one of the help that she's going down to the boat house, and I listen to her keep walking and now I don't hear her any more.

June 14, 1938, Tuesday
I know it's been a long time since I've written anything. They haven't let me sleep in my room, now that Aunt Rose is staying here. And nobody will let me be by myself, so I don't have any time to write. I heard Father telling one of the help that Aunt Rose would be here until after the funeral, which I guess is going to be Saturday. And then I'll get my room back, but it seems like even then that everybody is going to be watching me. And I heard somebody say that Mother and I had been arguing but it wasn't my fault, that even if Mother hadn't been diverted that she still might have flipped the boat and hit her head, but nobody knows how it happened because nobody saw it. But I didn't want to hear any more and I was able to sneak into my room and now I'm writing this. But I've got to go back downstairs, even though I wish I could just stay up here.

June 18, 1938, Saturday
We went out to the cemetery today. I rode down with Aunt Rose on one side of me and Father on the other, but I wanted to sit by the window like Jed and look outside.

I've been to the cemetery before, to see Grandma Jane's funeral, and to visit Grandpa Harper's grave a couple of times. This time, it's like we had the whole yard to ourselves, and I looked up at all the old crosses, and the statues, with the angels that looked down on me with their calm sad stone faces. And this one angel kind of scared me a little, and I kept looking up at him while they were reading the prayers for Mother, and it almost seemed like he might swoop down on us and carry Mother away in her coffin. And Father pulled my arm and made me look down, since I was the only one looking up, but I could feel the angel even while I looked at the square hole in between the rocks.

June 19, 1938, Sunday
We will be heading to church soon, but I needed to write about the most extraordinary thing! Aunt Rose left yesterday afternoon, and Father said that I could sleep in my room alone, if it was OK with me. So after supper, I was lying in front of the fire and Father was listening to his radio, and the Opry was on WSM, and there were a bunch of corny jokes and the country music. I started to think about why I haven't cried since the accident, and I know that everybody keeps waiting for me to cry, but all I can think about is how I wish I could fly away from here. I know that I have a true home somewhere, and it's always light, and it never gets white hot like it does here, and I can listen to beautiful music and all I would have to do is fly, just like the lightning bugs, who carry their own silvery white light just like stars. And I had been telling Jed for the last couple of days that my home is in a silvery white palace like the stars, and when I went to bed last night I dreamed that the lightning bugs came to talk to me. But I don't remember all they said. All I remember is that it was the first time I've felt home since the start of this terrible summer, and I know that my home is out there somewhere, in my dreams.

1958
April 5, 1958, Saturday
Cocktail party, as noted. Last night at the Jackson house, down by the parish hall. Bill had gotten home early, since he wanted to shave again and wash the car-"have to make an impression," as usual. Incumbent on me to wear my white gloves and matching hat, since Bill says that it's my classiest look. I wonder: why exactly do we seek Annabelle Jackson's favor, when it's apparent to all that her recent bout of good fortune has more to do with the mere happenstance of a family home lying on property directly along the most expedient route of a newly-planned interstate. Certainly, it's not her husband's middling presidency of a university that fails year after year to unseat its old

professors, or to clear its library of outmoded and useless ideas. Nevertheless.

I laid out his clothes on the bed; he washed the Oldsmobile; Sarah got home from her friend's after school and I left the Jacksons' number for Bernadette-thank goodness she agreed to stay late last night and watch Sarah! Reminder: we need to give Bernadette an evening off, in reward.

Headed crosstown during sunset. Bill a little tense—started trying to make conversation. He brought up our wedding day, out of nowhere. He started to say how it was the happiest day of his life, with the possible exception of the day that Sarah was born. Then he started talking about how Sarah was going to be married some day, and we should start saving now to make sure that she had a proper wedding, like the one we'd had. And how much he'd liked Father, and how it was obviously important to Father that his daughter was brought up right, and that he'd spared no expense on our wedding. I was listening in a state of semi-bemusement. Tried to soothe him-not like him to be this nervous. (He does seem much better this morning, though.)

Finally, we arrived at the Jackson house. Already quite crowded; clearly, none of Annabelle Jackson's friends throughout the parish declined THIS invitation! Lost Bill to the room where the men stand and talk; found myself in the dining room with three small groups of women, and a very few men who seemed ill at ease. One was some local professor, a young man who had just returned from a trip to the South Pacific, where he had been studying some tribe or another, gathering materials for a course he's preparing to teach this fall. Pretty interesting-very much what we read in the National Geographic. We were making small talk when Annabelle marched in, followed by a maid with a tray of hors d'oeuvres, and announced that she could never live in a country so savage that they didn't have iceboxes. We all agreed, and Annabelle went on to ask him if they had churches over there. He said that there had been some attempt to Christianize the island-evidently, mostly unsuccessful. It seems that their community doesn't need one more reason to gather, since they see each other all day all the time anyway. All in all, an interesting conversation.

Bill asked me about it on the drive home. Didn't remember the particulars. I guess Jed was at the party, even though I didn't see him—Bill said they'd been talking, and that Jed was all excited about something. I'll have to give him a call this week.

April 8, 1958, Tuesday
Just got off the phone with Jed. He's selling Anderson House.

He said he could get a huge price, almost double market-value, from the Army Corps of Engineers-part of that damned Gulf Outlet that everybody's always talking about. They can level the house and bore some sort of waterway through there. But Anderson House! My beautiful old house! I was quite blunt; I said that a house built and maintained in the proper Southern style has a memory, and that the Gulf Outlet will come along and obliterate our land and drain away our pool of memories until we might as well forget our world, our way of life. He DARED tell me that I've always been too melodramatic! And anyway, he has been wanting to move away to Atlanta, and this is his opportunity and he's taking it.

April 10, 1958, Thursday
Bill is away at work, and Sarah is at school. I've spent part of my morning talking to Jed, and it seems he's moving more quickly than I realized. He had surveyors from the Army out this morning, and it seems that only formalities stand between and the final sale. He has contacts with an advertising firm that has an office in Atlanta. It won't be long.

I had the strangest dream last night. I was in the midst of one of those game shows that you see on television-in fact, it was just like being within a television show, like "21." The host was asking me questions, and I was watching his mouth because I found it hard to understand what he was asking me. Nevertheless, I was able to answer correctly, because my mouth would open and words would come out and I would wait to see if they were the correct ones, and then the audience would cheer and I was overwhelmed by a feeling of relief. But finally, he asked me one question, and no matter how many times I weighed it in my mind, I couldn't understand it. So I asked him to repeat it, and he glared at me with a cold hostility before mouthing, slowly, "What was the morning of the world?" What a peculiar question! And so I kept considering it, wondering whatever it could possibly mean. "What was the morning of the world?"

And it occurred to me that I was supposed to answer that it was my wedding day. And the more I thought about it, the more I realized that it was the correct answer-my wedding day is the morning of my world. And yet, I realized that it was also the most base lie. And just as I discerned this, I realized that I had already formed the words. I couldn't call back the lie that had been extracted from me.

2000
November 22, 2000, Wednesday
Tomorrow is a big day. Louise and Emily are planning to take me out, to the cemetery. I guess the official reason is that I need to pick out my spot within the family plot. Clearly, nobody has any illusions about how long I have left! And yet, I have to say that I'm eager to go. I don't get to wander out much; I don't have the strength to wheel myself around, and Louise is too busy with her job to cart her old grandma around, and Emily has school. But the main reason I'm looking forward to it is the last chance to see Mother, and Father, and dear Bill, before I join them.

Louise and Emily came by on Sunday, and we all started to argue about the past. Louise said that we need to move on and let the past be. I said that the past is always living on. Emily asked if I meant that I believe in ghosts, and I said, "Indeed I do!" I always have, I guess. So I asked them what they think. Emily piped in that the past is gone. She was talking about this court case that says we should pay back all the descendants of slaves, for all the harm we've caused, and she said that she wasn't even alive during the Civil War and that it's not her responsibility. And I said that the Civil War is still going on, and then I saw them look at each other, with that "our poor old crazy grandma" look that always makes me want to laugh. And I said, "Look at this world all around you. It all comes from somewhere-it's the way it is because of what it once was. Look at Atlanta-it's built the way it is because it got burnt down in the Civil War-go out past the east side of town until you come to the Gulf Outlet-the swamps out there are dying because of all the salt water that the Gulf Outlet is letting into them. Everything you do has a piece of the past in it. I believe in ghosts! They're all around you. They're not a bunch of spooks, wearing bed sheets, but they're still there, just the same; they're in the world you live in, and the air that you breathe."

November 23, 2000, Thursday
It's morning, and I'm waiting for the girls to come get me. I guess I'm going on my very own ghost hunt!

Just got back from Thanksgiving dinner. The girls wheeled me into my room and said goodnight, but I don't feel like going to sleep just yet. I want to write down what I'm feeling before I forget it.

When we got to the cemetery, Sarah's Bill lifted me out of the car and set me into my chair. The girls each grabbed a handle and started guiding me forward. I was looking up at all the angels, just like I was a little girl again. They've got the same ones they had back then; a few have lost a nose or an arm, and I remember the one that got demolished in '62. And there are a few more graves, although a bunch were relocated when they opened that new cemetery on the west side.

I was expecting that things wouldn't have changed much since the last time I was here-it was just this May, after all. And I guess they haven't. But it was different, nonetheless. In May, I looked at each grave and thought how long it had been: Mother, 62 years; Father, 50 years; Bill, 7 years. And I counted the years that I had gotten to spend with each of them, and thought about the eternity I was about to, and for some reason the years seemed to outweigh the eternity. But it was different. This time, I found myself looking at the ground, thinking about how long it had been there, and the stone for the vaults and the headstones and the angels. What piece of rock went about its millions of years, essentially unchanged, before some grave artist cut it up and found an angel trapped, unbreathing, inside it? Did the rock think that the grave artist was going to take responsibility for its upkeep? Or did it know that the grave artist figured his work was done when he put the chisel down?

And yet those millions of years as a rock, and the hundred or so years it gets to be an angel before somebody decides it's too ugly to live and grinds it into a fine dust and adds it to a sidewalk somewhere-all that is nothing compared to what I've got to come. And then I thought, "What an unworthy creature I am! I'm in the presence of the bodies of my family for the last earthly time, and I'm grieved over some unfeeling piece of rock instead of tending to the matter at hand!" I started to shake a little, and when I stopped everybody was looking at me, so I had the girls push me over to the place that we all assumed was going to be mine. And we all agreed that it was a fine spot, next to Bill and at the foot of Mother and Father, and we talked for a few more minutes and then they took me off to dinner.

November 25, 2000, Saturday
I had one of my peculiar dreams last night. I dreamt that I stood out on a veranda with the curtains pulling at each side of me, and I was watching the morning start itself up and I suddenly had this feeling that I couldn't quite decipher. And so I started doing what I always do: took an inventory of each part of my body, trying to see which one was ailing me right now. And I realized that the feeling was coming from my mouth-there was a dull aching, and an unpleasant sense of little things going on inside that big space. Suddenly it was quite clear: there were a bunch of little hard things running around in there, and I would seek them out with my tongue and feel them race past and I was almost chewing. It took me a second or two further before I finally realized what they were: my teeth had come back! Come back to haunt me, I guess. Anyway, they were trying to find their old spots and root themselves back in, but they were all confused and kept playing musical chairs in my mouth, and I finally got tired of it and started spitting them out. But each time I thought I'd spat the last one, I'd root around with my tongue and find another hiding in some crevice or another and have to lure it out too. Teeth! I'm done with you, that's for sure.

November 26, 2000, Sunday
Been doing a bit of thinking. Sarah tells me that the girls spend a lot of time on this Internet, and that you can find anything there: old music and pictures of how things used to be, and I guess she thought that it might interest me. And you know, it did for a while, but I'm thinking now that the way things used to look is just not my concern.

I'm not going back on anything I've said-the ghosts are here, and people had better get used to seeing them. But I don't need them, because I'm not doing anything with them. I'm not planning some waterway, so I don't need the ghost of an old house to tell me not to poison a swamp and kill a bunch of birds. I'm not cutting up any old rocks, so I don't have the ruin of some old angel on my conscience. Somebody else (I guess I mean somebody YOUNG) has got to remember everything, to keep his conscience clean. All in all, I'd have to

say that my conscience is pretty clean.

I don't know where Mother's soul is now; if it's in Heaven, or in a coffin, or at the bottom of a lake. But wherever it is, I suspect it's not wasting any time worrying about me. It's probably just resting.

Well, now it's my time to rest.

Evaluation
Did we succeed?

We did indeed sacrifice "one of the basic rewards of storytelling." We realized that the most obvious aspect of our installation was our user-interface, not our narrative, and our user-interface is much more likely to provide a fragmentary experience than a cohesive one. (In fact, that's one of its purposes: to see if people cooperate enough to forge a cohesive experience.) We realized that our work was going to get scrambled; the images that we worked so hard on would be seen in passing, or sped up, or looped; that transitions we spent days to perfect, or lines we worked on again and again to make sure they could be heard clearly, would be cut up, interrupted, mangled.

And so we swallowed hard and proceeded. One of the values that we brought into the process was that we don't own the art work, or Emily's story; once it leaves our hands, it's up to the participants in the installation to provide meaning, not us. This isn't literary theory (what's commonly referred to as affective, or "reader-response," criticism); this is a fundamental element of our user-interface. Did we provide an effective narrative experience? We certainly didn't provide the one we started with, but we feel that some sort of narrative, however fragmentary, took place. Because you provided it.

Installation Design

Overview Inspired by installations at Siggraph and the "Swing" installation at LACMA by Jennifer Steinkamp and Jimmy Johnson, where users interacted with digital media by swinging, we decided to create a space where interaction with digital media was 'fun' rather than conventional.

In keeping with our subject of dreams, we designed an installation based upon a canopy bed. A sensor matrix will be placed in the bed and users will interact with the space by triggering these sensors. This internal environment will be separated by a curtain. It will contain imagery illustrating events from the central character's life and the dreams they trigger. Users in the external environment will be able to control the information (aural and visual) that the user on the bed experiences. An LCD projector will project the various director dream sequences onto the canopy and another will project them onto a screen located in the external environment. The canopy will act as a screen. Users on the bed will be able to view the different dreams on the canopy screen and become part of the dreams themselves. These users will also be able to subtly change the color, speed, or mood of the dream by interacting with the sensor matrix on the bed.

The Two Environments
We are creating two sub-environments within the main space. The user interactions in one environment will affect the experience of other users in the other environment.

Rules that govern the user interaction: The results of the interaction must have a meaning or a correlation with the action that is being performed. For example: movement on the bed may cause a change on the movie pace- the movie will play faster. The number of people interacting at the same time will be a parameter to explore (one user will produce a result that is different from two or three users). Results from the user interaction: the properties of the movie that is projected onto the bed, and/or sound environment created by the speakers, will change. Different properties will be considered: pace, rhythm, volume, colors and overlays.

Internal Environment
User interaction: the users trigger events by interacting with the bed. Different interactions will be considered. For example: lying still or rolling from side to side. The effect of these user interactions with the bed will be projected simultaneously onto the bed's canopy and into the external environment. The Canopy Bed

The dream sequences triggered by user interaction in the external environment will be projected onto the canopy of the bed.

Our decision to use a bed as the interactive object in the internal environment is based on the following assumptions:

An environment which simulates the natural sleep/dream environment will facilitate user immersion.

Most people will adopt the standard position of lying on their backs and looking up at the canopy. This should be a comfortable position for them and will further assist immersion in the environment.

The dream sequences will take on an additional 'dream-like' quality due to the nature of the canopy screen. Movement in the room, air currents, etc. will affect the canopy, causing it to flutter. In other words, this will not be a static screen, more movement and free-floating. (need to fix this)

Although this is environment is natural and unthreatening in a private setting, in a public setting some people may find it unsettling and feel too vulnerable. It will be interesting to see how people's interaction with this environment are influenced by their inhibitions.

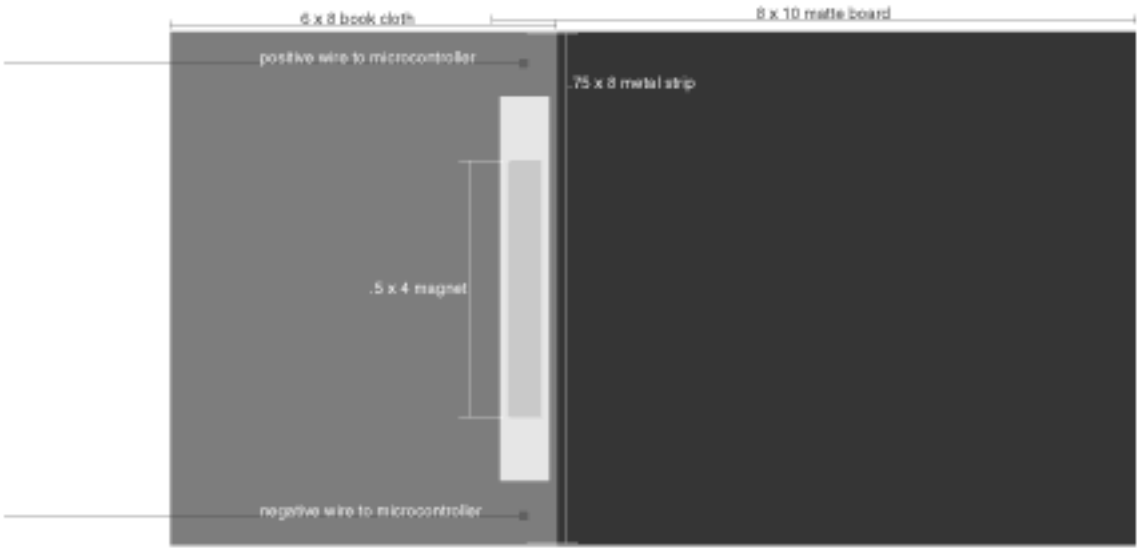
External Environment
User interaction: the user triggers events by interacting with different objects located at the media stations. The sound object will set the mood of the dream. The visual object will set the life phase of the central character and the specific event trigger. By interacting with these objects, the EUs determine the dreams the IU experiences. The media will combine to create either pleasant dreams or nightmares depending upon the choices the EUs make. The EU has priority over the IU. The IU can change the properties of his dreams by interacting with the bed. The curtain separates the two environments, reinforcing the division between internal and external environment.

The effects of these different users interactions will be projected onto a screen located in front of the media stations.

Media Stations/Interactive Objects
Visual Object: Interactive Photo Album

Sets the life phase of the central character and specific event trigger.

Conceptualization Process
Our group was inspired by the Spress group of the previous year, particularly their light sensor book. Initially we played with ideas incorporating a light sensor as the trigger mechanism but after realizing that difficulties calibrating the light sensors along with practical structural limitations caused us to discard this line of thinking.



Media Station: Visual Object: Interactive Photo Album
First Design

The solution as a simple switch mechanism.

We researched various binding techniques. Our desire was to create an interactive book that was not only visually interesting, but durable enough to withstand high use.

Main issues/problems
Hiding the wires
Making sure the pages made contact
Making sure that the wires were protected to avoid breakage
Making sure that the wires were separated to avoid crossed wires
Creating a program
Making sure that the book was visually interesting and fit with the overall installation

Solution
The following is a schematic of our original book design.

Discoveries
Our first effort was not successful due to the following:

We stripped the plastic protective coating off the wires in order to ensure that they added as little bulk to the binding as possible. However, by doing so we experienced breakages and crossed wires, both causing pages to fail.

After adding the photo corners and photos the pages no longer made contact. The solution was the addition of thin magnets to each page to insure contact was made. This was highly successful.

Initial problems with the book stemmed from electrical engineering inexperience and inconsistencies in both wiring and putting together the pages. Although we explored other options to the basic format of the book, we discovered by making the following adjustments to our first model the problems initially experienced were solved.

Developing a system whereas the positive wire was attached to the front, top of the page and the negative wire was attached to the back, bottom of the page.

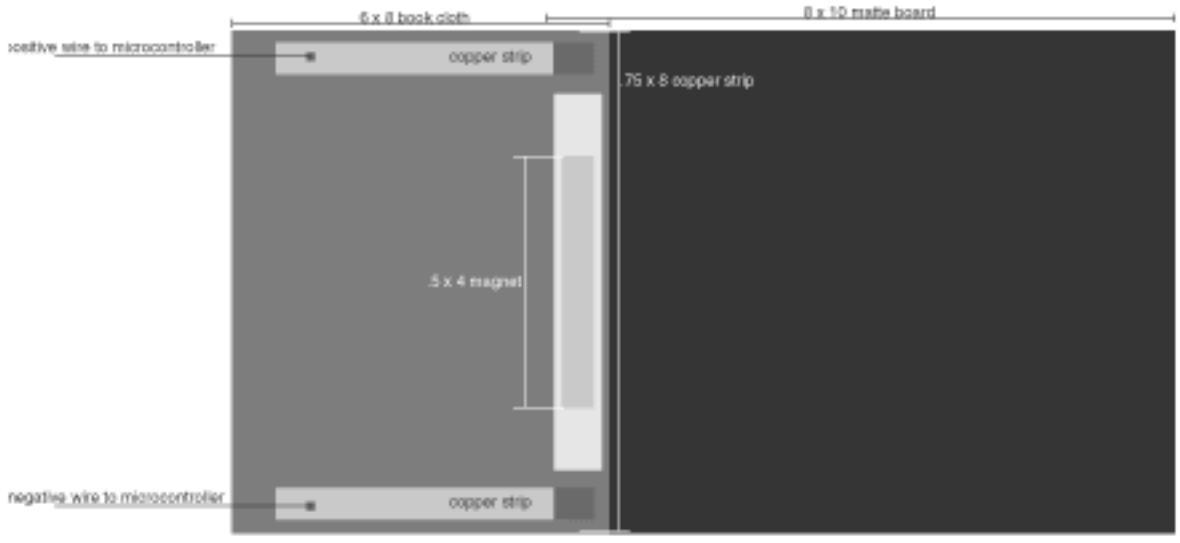
Choosing specific colors for the positive and negative wires (ex. positive = red, negative = black)

Keeping intact the plastic coating on the wires.

Adding a thin copper strip that was soldered to the contact strip and provided extra structural support. The also prevented the wire breakages caused by constant page turning.

Improving our basic understanding of electrical wiring and testing procedures.

Book Function
The book itself sets both the phase in the central character's life and the specific event which acts as a trigger for the dream sequence. We chose universal events/subjects in order to facilitate user understanding and association.



Media Station: Visual Object: Interactive Photo Album
Final Design

Title: Dream**Life** 06.01 Medium: Installation Dimensions: 27ft x 39ft Date: June 2001

The 9 subject/event areas are and the phases that they represent are:
Mother (childhood)
Child (childhood)
Father (childhood)
Marriage (adult)
Birth (adult)
Change (adult)
Acceptance (old age)
Peace (old age)
Death (old age)

As the participant flips through the pages of the book, s/he triggers the different dream sequences that are projected onto a screen located in front of the media stations in the external environment and onto the canopy of the bed in the internal environment. (insert relevant lingo info regarding timing, lags etc.)

Sound Object: Radio - See Sensor Section below
Curtain
Acting as both a physical and symbolic separation between the internal and external environments is a plastic pocket curtain. One side of the curtain contains imagery of the central character's external/waking reality and the other side contains imagery associated with her dream life. Significant people, places and/or events are printed on translucent paper to visually convey the experience of how aspects of our everyday life cross into our dreams.

Sensors.

Sensors are the core of any interactive space; they are necessary to gain user input and transform that input so that a cause and effect concept is evident to the user. Wanting to create a multi-user, multi-layered interactive environment meant implementing sensors to acquire user input and, in turn, manipulate presented (projected) media. Our goal is to create a seamless, well-integrated and transparent series of UIs in the space. As discussed in chapter 6 (Installation Design) these interfaces occupy two distinct but inter-related parts of DreamLife we call the internal and external environments

Sensors in the External Space
User input from the external space is captured and controlled by two interactive media devices or "media stations." Both devices suggest real possessions of the narrative's main character, one being an old radio (circa 1938) and the other a book in the form of a photo album or pictorial diary.

The Book
Figure 7.1 - The Book in action

One media station we've created is an interactive book. This book, however, assumes the form of a pictorial diary with one or more images representing an event from the narrative on each page.

The book generates its interactivity through the use of a switch array, each page simply having a contact point with its neighbor page. If any of these circuits are broken, i.e. when a page is displayed and only one can be displayed at a time), the circuit associated with that page is closed and software triggers a "page (x)" message which is sent to our Director movie. As outlined in our section on Lingo code, each page has a given place in the hierarchical structure in the environment and it's associated media is either called up or queued for display via projection.

Book - Physical Description
The book's physical characteristics are as follows:

The Radio
The sound media station is built within the housing of an actual antique radio (circa 1938). It is a Viking Radio, model number 49-33. It's wooden case is in exceptional condition for it's age, which is the primary reason for purchasing this particular unit. It was purchased on eBay (www.ebay.com) for $18.00 plus $10 shipping.

The radio was purchased in March 2001 in working condition from a private party in Manitoba, Canada. It did not have a face-plate. Essentially, the radio has been cleared of most of it's original electronic components to make way for wiring and a switch array secured to the under part of the tuning plate of the chassis (see figure 7.2E)

7.2A - The sound station as it appeared in the DreamLife installation, June 6, 2001

7.2B - A front view of the radio as it appeared immediately after purchase in April 2001. In this photo, the knobs had been removed. Notice the missing dial face which is how the radio was shipped.

7.2C - A rear view of the radio shell with the chassis removed. The wire lead coming from the bottom/front is connected to the IR Module attached on the underside of the shell.

7.2D - The chassis from the front. The tuning knob is still attached. Notice the thin cable and tuning cable wheel (on the right side) which formed the tuning mechanism. Also note the tuning needle itself on top of the top cross plate of the chassis. The ground wire was soldered to the underside of this in order to make contact with metal hanging from the underside of the cross plate.

7.2E - A shot of the BasicStamp and BOE with a rear view of the chassis and shell. Eventually, the BOE was positioned under the chassis where the tubes are mounted. Notice the red wires coming from the switch array mounted on the underside of the top cross plate.

Radio - Physical Description.
The radio's physical characteristics are as follows:

sensor type(s)

1) contact switch array (small pieces of shim metal and copper wire)
2) 560 Ohm potentiometer (original component)
3) Infrared Module (Sharp GP2D12 Infrared Ranger.)

The Mood Knob.
Originally the tuning knob, this apparatus now guides a piece of 18 gauge insulated copper wire (.5" exposed copper lead at end) along the switch array. The switch array has been fitted to the under part of the tuning plate of the chassis. Shreds of .0025" thick shim steel has been soldered to copper leads and hang down from the underside of the tuning plate of the chassis. As the knob is turned, the copper lead, soldered to the underside of the tuning needle, passes from one piece of shim steel to the next. When the copper lead and the shim steel make contact, the corresponding circuit is closed and +5v is sent to that circuit's input pin on the Basic Stamp. As the shim steel is flexible, it travels with the copper wire as it passes thus simulating the distance in the revolution of the knob that a radio station would occupy. There are nine contact switches and therefore nine virtual radio stations. These stations will access different sound files (MP3s) depending upon which page of the book is currently being viewed.

The switch array for the radio is similar to the book's switch array and we have therefore used a modified version of the book's code for our switch array on the radio.

The Ambient Knob
The left knob, originally the power/volume knob, is a 560-ohm potentiometer (i.e. non-linear). We are using this original electronic component and supplying +5v to it. In turn, the wiper terminal sends a voltage of variable resistance to the Vin pin on an ADC 0831, an 8-bit single-channel analog-to-digital integrated circuit.

From the DO (Direct Out) pin on the ADC 0831, we get 8-bit values sent from 0 to 255 through the computer's serial port where a Serial Xtra captures and manages this data for Lingo processing. Subsequently, a Lingo script maps the 0-255 8-bit value to a scale of 1-100. This decimal scale is broken into 10 equal value-segments, each calling a specific ambient file.

Infrared Detection
Figure 7.3 - The Sharp GP2D12 Infrared Ranger

One interactive element that is not processed through Director, Lingo or the Multi-user server is the infrared (IR) detection system. This sub-system simply senses presence in front of the sound station (radio) and triggers two ultra-bright LEDs mounted on the inside of the radio chassis. This gives a "smart" and almost eerie quality to the radio when approached. For this application, we are using a Sharp GP2D12 Infrared Ranger. More information on this IR module can be obtained from http://www.acroname.com/robotics/parts/R48-IR12.html.

Sensors in the Internal Space
Note: The canopy bed, its sensors and the hardware for displaying media (visual and aural) constitutes and defines our internal space.The canopy bed, as outlined in the installation design section of this document, is the vehicle by which users will experience the projected media of the internal space. The bed is fitted with six active Force-sensing Resistors (FSRs) which deliver a variable voltage in proportion to the amount of force exerted. Three leads are attached to each FSR: power (+5v), ground and signal. The signal lead interfaces directly with a channel on an ADC 0838 (see below) and in turn each channel is read by code (also below) running on the Basic Stamp.

Force-Sensing Resistors (FSRs).
Typically, Force-Sensing Resistors (FSRs) were developed in the early 1980s by instrument (keyboard and drum machine) manufacturers

in the pursuit of creating touch-sensitive keyboards and drumpad surfaces as well as implementing after-touch mechanisms. More recently, many automotive engineers have come to regard FSRs as the “building blocks” of today’s intelligent airbag detection systems. Generally speaking, an FSR is a combination of conductive and non-conductive materials designed so that, when pressure is exerted upon the sensor, the result is a variable voltage relative to the supply voltage and the amount of force exerted.

FSRs are commercially available. Interlink is a manufacturer of a wide variety of FSRs and devices that implement FSR technology such as touch pads and various pressure pointing technologies. Another company, Measurement Specialties Inc. utilize piezo-film to create FSRs. However, being dual-coupled, these sensors are more difficult to control and exhibit far more volatile pressure-voltage curves.

Physical Description of Our FSRs
The FSRs we have decided to use are on loan from composer, performer and CSUH staff member Scot Gresham-Lancaster. Having worked with FSRs in a wide variety of forms and for a wide variety of end uses, Mr. Gresham-Lancaster proved to be a gold mine of valuable assistance, not only materially but intellectually. Left over from an old project were nine 8”x6” FSRs, each having six connector pins (three per side of sensor - power, gnd and signal). Each sensor, being identical, has a “comb” pattern of conductive material sandwiched between two sheets of Kylar.

Mounting
Due to the fragility of the sensors, we have taken steps to protect them against excessive wear and tear in the course of user interaction. Therefore, we have positioned three 1’ x 6’ strips of .25” thick clear vinyl over each column of sensors. This material is supple enough to allow a somewhat accurate pressure reading (but not without adding stress values!) yet durable enough to take most of the stress of user interaction without showing any signs of wear.

The sensors themselves are mounted onto a hard cardboard surface which is then fixed to the top of a full-size box-spring (without mattress). We can rely on the “sticky” nature of the vinyl to keep the FSRs in place in conjunction with a modest amount of duct tape.

Over the vinyl, we have positioned a full-size comforter onto which our user(s) directly lie. Figure 7.4 shows the dimensions of the bed and the locations of the sensors on the bed.

Figure 7.4 Schematic for the mounting of our FSRs

Electronic Circuits and Electronic Components
Our final solution for the bed and the book couldn't be simpler, however it took much experimenting to gain simplicity. The sensors are connected directly to Parallax’s Board of Education. The circuits are designed to either 1) reduce input voltages from sensors (to not more than 5v), 2) interface various ICs, exclusively ADC chips for analog-digital conversion and 3) signal conditioning (IR module on radio).

Figure 7.5 The ADC 0838 mounted on Parallax Inc.'s Board of Education. Notice Channel 1 at the top left of the IC is being used but the rest of the channels below it are un-occupied in this photo.

Figure 7.6 The FSR and ADC interface for the bed.

Of critical importance for more than one interactive station in the space is the ability to convert analog sensor data to digital data. This is accomplished by using a multichannel ADC0838 (8-bit eight-channel analog to digital converter - see figure 7.5 and 7.6 above) to interface with the FSRs, which deliver analog variable voltage, on the bed. Furthermore, we use an ADC0831 (a single channel version of the 0838) to interface with the potentiometer on the radio (left knob - Ambient). The nature of a switch being binary, the switch-based sensors (radio tuning knob) and all the pages of the book do not require analog to digital conversion (see figure 7.7 below).

Figure 7.7 The Book circuit showing the simplicity of the switch array.

The circuit for the IR module requires a transistor to boost the signal to increase sensitivity of the module. Testing the module without any signal conditioning showed a poor or non-existent response from a distance of just six inches. As this needed to detect presence of a person as they approached the radio, we required a detection distance of at least twelve inches minimum. With the implementation of the transistor/capacitor circuit with the IR, we achieved this range consistently.

Bed Sensor Development
Homemade FSRs
Before coming across Scot Gresham-Lancaster’s early Interlink FSRs in early November 2000, the sensor team built a number of “homemade” FSRs using anti-static foam (used largely for protecting pins of ICs in retail packaging) sandwiched between two copper-plated PC boards. To each PC board was soldered a wire lead that ran directly to a pressure-to-voltage circuit (see below). As the PC boards were too rigid to use on a bed underneath a user and would therefore not be “transparent,” we experimented with using varieties of aluminum and shim steel sheets as conductive materials within which anti-static foam was encased. We found that the main problem using any of this type of FSR centered around the unreliable characteristics of the anti-static foam. It had a tendency to compress over time and therefore

change resistive properties with even a slight amount of use.

The Early Circuits and the Components
Figure 7.6 The 741 Op Amp IC and voltage divider. An early pressure-to-voltage circuit the group used but discarded for our final simpler arrangement.

The early circuits we used to test our pressure-to-voltage sensors were primarily based upon the schematic found on p. 31 in Radio Shack's Electronic Sensors Circuits and Projects (Forest Mims III, Radio Shack 1998-2000).Ê In order to acquire ample signal from these homemade devices, we experimented with a wide variety of resistor values between pin 6 and 2 of a 741 Op Amp IC.Ê Later, while experimenting with peizo-film, we used the same circuit to condition and boost signal, as well as an RC (resistor-capacitor) circuit which would allow a capacitor to store and discharge voltage at varying degrees.

Piezo-film Pressure sensors
Piezo film is a AC-coupled, PVC material which is highly sensitive to strain and impact. However, due to their volatility and high level of sensitivity, we were not able to make piezo-film sensors work as satisfactorily as the standard FSRs. Essentially, the problem with piezo film was that the sensor would jump to +5v with very little weight or exertion. Obviously, with our application involving total adult body weight, this was clearly unacceptable. We experimented with voltage divider circuits, RC circuits (resistor-capacitor circuits which were necessary to condition the piezo signal being AC coupled) and a wide variety of methods to achieve acceptable results. We came close using one of piezo sensor (8" x 8") manufactured for DreamLife 06.01 by Measurement Specialties Inc (MSI). This particular sensor, part of a group of four sent to us in March of 2001 from MSI as prototypes for testing tended to achieve a smoother pressure/voltage curve. Still, the Interlink FSRs, in spite of their apparent fragility, were the better choice for the bed.

Landmarks
October 2000 - Primary obstacle was to develop a sensor to detect movement on a bed. We broke up the dimensions of the bed into a grid to formulate "hot spots" each of which would produce a different effect upon the internal environment.

November 2000 - A critical hurdle was passed on Wednesday, Nov. 22nd when the sensor team (Mark Anderson and Marilenis Olivera) and David Johnson got an FSR (Force-sensing resistor) to send variable voltage through an analog to digital converter (AD0831) which in turn sent the signal through a parallel cable to the serial port of a PC. At that point, we implemented a Serial Xtra which read the binary value coming from the serial port on the PC.

December 6, 2000 - We passed another landmark by assembling a prototype system which included the subsystem described above with four FSRs mapped to the following parameters: Audio Loop and Volume, Opacity and Text file swapping.

November 2000 - In this presentation we integrated the sensors with director movies on networked machines via Shockwave Multiuser Server, a media file database-mining algorithm (lingo) and projection. This system was presented to the Multimedia Graduate Forum on Wednesday December 6, 2000.

December 2000 - The Book and the Bed were integrated into one BasicStamp and BOE (Board of Education - see February 2001 - www.parallaxinc.com for more information). This sensor-driven pair was successfully integrated into the overall MultiUserServer architecture network.

April 2001 - Discarded Sensor/ICube/MAX interactive sub-system for the Radio (sound station). As attractive and sophisticated as this system was, it simply posed too many variables and complications. It was abandoned for a simpler switch array (see section 7.1.2 above).

May 2001 - The Book, Radio and Bed sensor arrays were put online and manipulated in real-time via the MultiUser Server network architecture simulating the finished installation design, complete with a full compliment of A/V hardware and projectors. We experienced problems with quality of media delivery and network reliability but subsequently debugged the system.

Using a Micro-controller - The Basic Stamp
The interface between sensor and computer is the micro-controller. We use it for converting analog to digital values, performing loops and essentially controlling all of the user I/O in the installation. Basically, it is our first "line of defense" for data processing. The data flow: sensor - electronic circuit - micro-controller - computer (via serial port) - third-party Xtra (Director) - Director (lingo). This section will effectively cover the second, third and fourth steps in this sequence.

Figure 7.7 - The Basic Stamp II

Our installation uses three BasicStamp IIs. One interfaces with the interactive bed, one with the book and a third with the radio. We have opted for a micro-controller for each interactive "station" for the following reasons:

Proximity - As each station is physically separated in the space, the degradation of serial data over distances of twelve feet prohibits using a

single stamp for any two or more stations.

Capacity - The number of inputs of each “station” requires a separate micro-controller. The radio utilizes fourteen of 16 I/O pins on the stamp itself while the book using nine and the bed six.

The Board of Education
Figure 7.8 Parallax’s Board of Education without mounted Basic Stamp 2

Parallax’s Board of Education is designed as an experimental breadboard platform for building circuits that interface with the Basic Stamp. Among other features, it provides voltage regulation (from a 9v source) to a steady 5v, easy access to the Stamp’s I/O pins, choice of AC adapter or 9v battery power source, a serial port for serial communication with a computer, and a breadboard adequate enough to build multiple circuits to interface with the Stamp. We have built all of our circuits which interface between sensor and microcontroller on these boards and have integrated three of them (one per stamp) in the installation.More information can be found at www.parallaxinc.com.

Programming the Stamp - PBASIC
Parallax’s proprietary form of BASIC used to program the Basic Stamp is called PBASIC. Our Code was created to accommodate the specific tasks and challenges of our interactive environment. Having three stamps, we have separate programs for the stamp controlling the radio, the bed and the book individually.

The Bed

```
‘Program to read input from the Interactive Canopy Bed
‘Created by: Marilenis Olivera
‘Winter 2001 - Multimedia Graduate Program
‘Schematic of the ADC0838 - Reads voltage through channels (analog inputs) 1-8 ‘and converts the voltage to a digital signal
‘Each sensor from the bed is connected to one of the input pins on
‘the ADC0838.
“ _____-
‘ * indicates an active low pin
‘Initializations for the ADC0838
ADCRes VAR BYTE ‘ A-to-D result: one byte.
CS CON 0 ‘ Chip select is pin 0.
ADCIn CON 1 ‘ Data input to ADC is pin 1.
ADCOut CON 2 ‘ Data output from ADC is pin 2.
CLK CON 3 ‘ Clock is pin 3.
Channel VAR BYTE ‘ Number of the channel we want to measure
InitBits VAR BYTE ‘ Sequence of bits for initialization
OUTA = %0001 ‘ Set Chip Select High to deselect ADC
DIRA = %1011 ‘ Set direction bits properly
‘BED Loop: go through channels 0 to 5 and output the result of ‘the conversion to the basic stamp
BED:
FOR Channel = 0 TO 5 ‘ Go through all the channels
LOW CS ‘ Activate the ADC0838.
‘Calculate initialization bits. Bit definitions are as follows:
‘Bits 7..5 = all 0’s (will be ignored by the ADC)
‘Bit 4 = 1 (Start bit)
‘Bit 3 = 1 (Single mode)
‘Bit 2 = Odd channel selector (bit 0 of channel #)
‘Bit 1 = Channel selector (bit 2 of channel #)
‘Bit 0 = Channel selector (bit 1 of channel #)
InitBits = %11000 | ((Channel & %001) << 2) | ((Channel & %110) >> 1)
‘Shift out the initialization bits
SHIFTOUT ADCIn,CLK,MSBFIRST,[InitBits\8]
‘Shift in the 8-bit data and ignore the first bit (dummy bit)
SHIFTIN ADCOut,CLK,MSBPOST,[ADCRes\9]
HIGH CS ‘ Deactivate the ADC0838.
‘ Show us the conversion result.
DEBUG “Channel “,DEC Channel, “: “,DEC ADCRes,CR
```

```
PAUSE 50 ' Wait a second.
NEXT ' Select next channel
GOTO BED
```

The Book

```
'Program to read input from the Interactive Book
'Created by: Marilenis Olivera
'Winter 2001 - Multimedia Graduate Program - CSUHayward
'Pins 1 to 9 in the Basic Stamp board are declared as input
'Each input corresponds to a page in the book
input 1
input 2
input 3
input 4
input 5
input 6
input 7
input 8
input 9
'Book Loop: Finds an open page in the book, it sends a message with the number of the page or with a 'of close book to the MU Server
'To check that a page is open or close, it checks the pin, if the value is 1 then the page is open, 'if is cero then the page is closed. 'This routine checks the pages from bottom to top, to avoid reading the pages that have been flipped already.
'Page 10 is send to the MU Server when the book is closed
```

BOOK:

```
if (in9 = 1) then PB9 'If 0 then Page 9 is closed with Page 3 else Page 4 is open
if (in8 = 1) then PB8 'If 0 then Page 8 is closed with Page 3 else Page 4 is open
if (in7 = 1) then PB7 'If 0 then Page 7 is closed with Page 3 else Page 4 is open
if (in6 = 1) then PB6 'If 0 then Page 6 is closed with Page 3 else Page 4 is open
if (in5 = 1) then PB5 'If 0 then Page 5 is closed with Page 3 else Page 4 is open
if (in4 = 1) then PB4 'If 0 then Page 4 is closed with Page 3 else Page 4 is open
if (in3 = 1) then PB3 'If 0 then Page 3 is closed with Page 3 else Page 4 is open
if (in2 = 1) then PB2 'If 0 then Page 2 is closed with Page 2 else Page 3 is open
if (in1 = 1) then PB1 'If 0 then Page 1 is closed with Page 1 else Page 2 is open
debug "Page 10", 13
Pause 100
goto BOOK
'PB1 to PB9 output a message to the debug window- Using the SerialXtra for director, all the output that is
'sent to the debug window is read by a director movie. This movie reads the book status and decides what movie
'should be projected by the cannopy projector.
PB1:
debug "Page 1", 13
Pause 100
goto BOOK
PB2:
debug "Page 2", 13
Pause 100
goto BOOK
PB3:
debug "Page 3", 13
Pause 100
goto BOOK
PB4:
debug "Page 4", 13
Pause 100
goto BOOK
PB5:
```

```
debug “Page 5”, 13
Pause 100
goto BOOK
PB6:
debug “Page 6”, 13
Pause 100
goto BOOK
PB7:
debug “Page 7”, 13
Pause 100
goto BOOK
PB8:
debug “Page 8”, 13
Pause 100
goto BOOK
PB9:
debug “Page 9”, 13
Pause 100
goto BOOK

findHeaderNum(“H4”);
```

The Radio

The Radio code is an integration of pre-existing PBASIC written for interfacing the BasicStamp II with the ADC 0831 A/D converter chip (see p.123 Ch.5 of Robotics v1.3 by ParallaxInc cited in our bibliography) and group-authored PBASIC from the book. The book code came in handy for checking the radio station switch array because they are very similar in nature. Finally, the IR module loop created by Tomonori Yamasaki which looped through the various routines of the radio code: “pot knob”, switch array or “radio loop”, and “IR Loop.” Originally, they radio was intended to have 12 stations. When a bug was encountered in mid-May, it was slightly scaled down to nine stations. Our final radio code was heavily modified by Tomonori Yamasaki (see figure 7.9) and is based on comparative loops, i.e. loops which essentially have two variables for each state or condition of each of the three sub-systems on the radio as outlined above. As it compares the states, it sends only values when the value has been changed. In other words, unlike the code for the bed and the book, the code for the radio sends data to the computer only when it has changed from its previous state. This seems more efficient and easier for Director and the Serial Xtra to handle, as opposed to handling a continuous stream of data as the book and the bed do.Furthermore, each station variable reserves only a single bit of memory space (see “st0 VAR stats1.BIT0” et seq below) which makes it processor and memory efficient as well.

Figure 7.9 Tomonori Yamasaki’s sketch of the algorithm used for the Radio Code

```
‘**************************Radio Code*******************************
‘****sensor input routines****
‘*IRLoop
‘*POT_Knob
‘*RadioLoop
‘*this is the sensor read routine runs as fast as possible
‘****send message routine****
‘*send_message
‘*this happens once in 12 sensor routine by default
‘***********************************************
declarations:
‘IR detection using two IR pairs on a BOE //p.123 Ch.5 of Robotics v1.3
IR_det var bit ‘declare IR_det as a variable and reserve a bit of memory
IR_detOld var bit
output 1 ‘set pin 1 as an output
HIGH 1 ‘set that output high +5
‘POTS!!!!- AMBIENT KNOB
‘ Program: ADC0831.BS2
‘ This program demonstrates the use of the BS2’s new Shiftin instruction
‘ for interfacing with the Microwire interface of the Nat’l Semiconductor
```

```
‘ ADC0831 8-bit analog-to-digital converter. It uses the same connections
‘ shown in the BS1 app note.

ADres var byte ‘ A-to-D result: one byte.
ADresOld var byte
CS con 2 ‘ Chip select is pin 0.
AData con 3 ‘ ADC data output is pin 1.
CLK con 4 ‘ Clock is pin 2.
high CS ‘ Deselect ADC to start.
stats1 VAR BYTE
stats2 VAR BYTE
oldStats1 VAR BYTE
oldStats2 VAR BYTE
temp VAR BYTE
temp2 VAR BYTE
st0 VAR stats1.BIT0
st1 VAR stats1.BIT1
st2 VAR stats1.BIT2
st3 VAR stats1.BIT3
st4 VAR stats1.BIT4
st5 VAR stats1.BIT5
st6 VAR stats1.BIT6
st7 VAR stats1.BIT7
st8 VAR stats2.BIT0
st9 VAR stats2.BIT1
st10 VAR stats2.BIT2
IR0 VAR stats2.BIT4

counta VAR BYTE
counta = 0
stats1 = 1
stats2 = 1
oldStats1 = 1
oldStats2 = 1
temp = 0
temp2 = 0
IR_det = 0
IR_detOld = 0
‘*****************************************************
IRLoop:
IR_det = in0 ‘for mark, 1 is high, 0 is low now.
if in0 = 0 then inverter0
if in0 = 1 then inverter1
inverter0:
IR_det = 0
goto exit
inverter1:
IR_det = 1
‘end conversion
exit:
goto POT_Knob
‘*************************************************************
‘ADC 0831 CODE
‘*************************************************************
POT_Knob:
low CS ‘ Activate the ADC0831.
shiftin AData,CLK,msbpost,[ADres\9] ‘ Shift in the data.
```

```
high CS ' Deactivate '0831.
goto RadioLoop ' Do it again.
'*********************************************************
'Radio Station Sensor input
'check 9 stations sensor
'every 12 times, transmit changed station message
'*********************************************************
input 6 'blah blah blah. inputs. [modified 5.17 for 9 stations]
input 7
input 8
input 9
input 10
input 11
input 12
input 13
input 14
RadioLoop: 'blah blah blah. read the ports.
st1 = in6
st2 = in7
st3 = in8
st4 = in9
st5 = in10
st6 = in11
st7 = in12
st8 = in13
st9 = in14
counta = counta + 1 'send read results after 12 times
if counta > 50 then SEND_MESSAGE 'checking all stations.
goto IRLoop 'go back to the top baby.
'*****************************************************
'SEND_MESSAGE routine
'this only happens sometimes a sec. 5 lines up, there's "counta > 12" thing, 12 is the number of loops
'sensor reads routine runs. if you want to reduce the amount of data, use bigger number, so you get less
'message.
'*****************************************************
SEND_MESSAGE:
temp = IR_detOld + IR_det 'changed? IR
if temp <> 1 then no_IR
debug "IR = ", bin1 IR_det, CR
if IR_det = 0 then led_on
led_off:
out1 = 0
goto exit_led
led_on:
out1 = 1
exit_led:
IR_detOld = IR_det
no_IR:
temp = AdresOld - Adres 'changed? Adres
if temp = 0 then no_Adres
debug ? Adres, 13
AdresOld = Adres
no_Adres:
counta = 0
temp = stats1 ^ oldStats1 'check with old result, see if changed. XOR
temp2 = stats2 ^ oldStats2
temp = temp | temp2 'get OR so the results added together.
```

```
if temp = 0 then IRLoop ‘ no change? go back to the top.

‘debug “changed!” ‘indicator for status change
‘debug bin stats1 ‘cont’d
oldStats1 = stats1 ‘store the new results.
oldStats2 = stats2


temp = stats1
‘debug “Value of temp=stats1 is “, bin temp, cr
temp = %11111111 - temp ‘reverse bits 11110111 -> 00001000
temp = NCD temp ‘returns the first 1 from left end (return is bit num)
‘debug “NCD temp “, dec temp, cr


temp2 = stats2 | %11111000 ‘you only need 3 channels (chan 9 - 11)
temp2 = stats2 | %11111100
‘debug “Value of temp2=stats2 is “, dec temp2, cr

temp2 = %11111111 - temp2
temp2 = NCD temp2
‘debug “NCD temp2 “, dec temp2, cr


‘debug dec temp
if temp <> 0 then stats1_send ‘change in 0-7?
if temp2 <> 0 then stats2_send ‘or change in 8-10?
goto noStat ‘then no change!
stats1_send:
goto send_stats
stats2_send:
temp = temp2 + 8 ‘make channel number... (high byte of word... if you like...)
send_stats
temp = temp - %00000001
debug “ST “, dec temp, cr ‘no need “cr”?
goto exitSTN
noStat:
debug “NO STATION SELECTED”, cr
exitSTN:
goto IRLoop
‘*****************************************************************
```

Software Architecture

NOTE: this chapter is written to document our code and to provide a knowledge base for other Macromedia developers. As such, at least an intermediate level of Lingo skill is assumed. There is no attempt to explain basic Lingo functionality or syntax.

Overview

Our software is written in Lingo, the scripting language of Macromedia Director. As of June 2001, we are using Director (ver. 8) to coordinate media among four major entities, which can be described conceptually as various Director movies:

The “Bed” Movie: this movie translates pressure from the sensors underneath the bed into Lingo messages, which it sends to the Canopy movie;

The “Book” Movie: this movie determines which page of the book is currently viewable and sends this information to the Canopy and Radio movies;

The “Radio” Movie: this movie receives input from the two knobs on the front of the radio, which it sends to the Canopy movie; it

responds as well to the page information it receives from the Book movie; and it uses both the knob and page information to play sounds in the external environment; and

The "Canopy" Movie: this movie receives input from the Bed, Book, and Radio movies, and uses this information to display aural and visual media in the internal environment.

The Bed, Book, and Radio movies all send messages to other movies; the Radio and Canopy movies receive messages and display media. These dataflows are diagrammed as follows:

Figure 1: Top-Level Architecture (all movies)

Note: These four movies each use the services of a fifth movie, the "Network" movie, to handle the mechanics of communicating with each other. The Network movie is not displayed as an entity in the top-level architecture because it does not initiate or receive messages; it merely serves as a conduit.

Lingo
Because we are using Macromedia Director as our platform, this allows us to use Director's scripting language, Lingo, as our primary language. We aren't using any pre-defined behaviors; all of the code is custom-written for this project. We are using Director 8.0, which gives us access to imaging Lingo, a series of functions that allow direct, pixel-level control of the screen, and to a more robust group of sound-control functions than were available in earlier versions of Lingo.

NOTE: there are several words that have specific meanings in the Director environment, which might cause some confusion in this context. Macromedia uses the word "movie" to describe a Director project and the word "projector" to describe the compiled executable that is created from a Director movie. In this chapter, we use the word "movie" in its Macromedia-specific sense, to describe a Director project on a single computer. However, we use the word "projector" to describe an overhead LCD projector, which projects our movie onto the bed's canopy. When we need to refer to a Macromedia projector, or to a different type of movie (such as a QuickTime movie), we will explicitly label it as such.

Shockwave Multiuser Server
A recent feature of Director is the Shockwave Multiuser Server, an application that allows Director movies on different computers to send messages to each other. The MU Server is an application that runs on one of the computers and directs message traffic among the various computer. The MU Server has a predefined format for these messages, which is described in depth below. Also, each of the movies instantiate the Multiuser Xtra, which provides the functions that allow the movie to send messages to the server.

Xtras
We employ several Xtras in our project:

Multiuser Xtra: each of the movies is bundled with the Multiuser Xtra, which provides the code that allows Lingo to send messages to the Shockwave Multiuser Server;

SerialXtra: the SerialXtra allows those movies that receive input from the Basic Stamp (i.e., the Bed, Book, and Radio movies) to read data from the serial port; and

File IO Xtra: the Network movie writes some of its data out to an external text file, to speed up initializing the movie. The reason for this is that one generic network movie is bundled with each of the four other movies, but the settings that are appropriate for the Radio movie are different from those that are appropriate for the Book movie (for example). Therefore, we provide text fields into which we can type the current settings, but we also want those settings available in case we need to do a very quick restart while the project is running. Writing those settings out to a text file, and then reading them in upon restarting, allows us to store those settings, so that they persist beyond a crash or a restart.

Version Tracking / Naming Conventions
The architecture described above is embodied in four Macromedia Director movies: Canopyx.dir, Bookx.dir, Bedx.dir, and Radiox.dir (the x indicates a version number, which is incremented with each major rev of a movie). Because each movie needs to be able to communicate with the other movies via the Shockwave Server, we have also created an additional movie, Networkx.dir, to handle the communication with the Server. This movie is included in the projector that is created from each of the movies.

At the time of writing, the revisions are Canopy4g.dir, Bed1a.dir, Book1a.dir, Radio5b.dir, and Network4.dir.

Each of these movies runs on a dedicated computer. In addition, one of the computers is also running the Shockwave Multiuser Server. In order to reduce overhead on either of the stations that outputs media, this server is currently installed on the computer running the Book movie.

Media Types
Director can handle a wide variety of media types. As of June 2001, we are using sound files (AIFF format) in the Canopy and Radio movies, and bitmapped images and QuickTime video as well in the Canopy movie. These types are manipulated extensively by our Lingo; in particular, bitmapped images are radically altered in the Canopy movie, using the Particle system engine described below.

General Object Structures
Common Structures in All Four Movies
While the Book, Bed, Canopy, and Radio movies all have different functions and approaches, they also share certain internal structural similarities. We deliberately use similar structures (for example, identically-named objects) for two reasons: to facilitate debugging, and to permit one generic Network movie to be bundled with each of the other movies, without having to customize the interface for each movie. These common structures are described in general terms here.

All of the movies make use of Lingo's object-oriented features (parent scripts, ancestors, property variables, and to a lesser extent behaviors). We make limited use of multiple child objects being spawned from the same parent script (one of OOP's distinct advantages) in the creation of our media lists (described below), but we also have many object constructors that create only one instance of an object. For this reason, we often refer to an object being "initialized," rather than "instantiated."

Instead of creating multiple instances, we employ the object-oriented features as an organizational tool, to allow explicit encapsulation of functionality, and to permit persistent variables without the inelegance of global variables. In particular, the encapsulation of functionality in objects is enforced across the four movies.

Top-Level Objects
This encapsulation has resulted in 5 high-level objects, which are the only global variables we use:

The Mailbox object (gMailboxObject), which handles communication between movies (on an abstract level) and storage of state information (such as the current page of the book);

The Logic object (gLogicObject), which decides what to do when there's a change in the system state;

The Database object (gDatabaseObject), which contains references to all the media that the movie can display (NOTE: this object only exists in the Canopy and Radio movies, since they're the only movies that display media);

The Display object (gDisplayObject), which handles the display of aural and visual media (NOTE: this object only exists in the Canopy and Radio movies, since they're the only movies that display media); and

The Server object (gMServer), which handles low-level communication between the movies. This object handles the mechanics of communicating with the Shockwave Server and is created by the Network movie.

Each object may contain many objects of its own as properties; for example, the Display object uses its own objects to handle the mechanics of displaying video, mixing audio, and running the Particle system. These "sub-objects" are not exposed to any of the other top-level objects-not even by accessor methods. All communication among the top-level objects occurs only at the top-level, and then it is up to the top-level to determine whether to pass a message down to a lower-level object. This can lead to redundancy (for example, a message from a Bed sensor is routed to the Display object, which in turn might send it to its Video Mixer object, which in turn sends it down to one of its Video Channel objects), but the benefit is in encapsulation and debugging.

In this way, the architecture of the objects is almost an inversion of an OSI stack: the top level provides communication services to the lower levels, so that messages originate at a lower level and then are routed up to the top. In addition, the communication among the objects is limited: only the Mailbox object talks to the Server object, for example-the other objects are not aware of it. This allows us to take advantage of polymorphism: the Network movie can be bundled separately with the other movies without modification, because it knows it needs to send messages to the Mailbox object, and doesn't care that the Mailbox object in the Book movie is different from the one in the Canopy movie so long as the interface is the same.

Communication Between Movies: Page, Mood, and Bed Sensors
There are two things the Canopy and Radio movies need to know in order to display media: which page of the Book is currently viewable, and what is the current mood of the piece (as determined by the Radio dial). In addition, the Canopy movie needs to know the current state of the sensors in the Bed, to determine how to display its media. The page, mood, and Bed sensor information is passed among the movies via the Shockwave Server in the following fashion:

Page: both the Canopy and Radio movies expect to receive an integer value, ranging from 1 - 10. Values from 1 - 9 indicate that pages 1 - 9 are currently viewable; a value of 10 means that the Book is closed. When a user changes pages in the Book, the Book movie sends a message to all other movies informing them of the new page. The Bed movie ignores the message; the Canopy and Radio movies respond

to it.

Mood: because there are 9 radio stations for every page of the book, the Radio movie keeps an internal value of 1 - 9 to describe the mood of the piece. (In general, lower numbers indicate a darker mood, while higher numbers indicate a happier mood, although we are far from rigorous in adhering to this!) When the user changes the radio station, the Radio movie sends a message to the Canopy movie that the overall mood has changed.

Bed Sensors: the array of Bed sensors measure weight on a scale from 0 - 255. When the value from a sensor changes, the Bed movie sends a message to the Canopy movie with the index and value for that particular sensor.

Lingo In Depth: The Canopy Movie
The Canopy movie is the primary output for the installation; it is the vehicle for our narrative and our visual media. The Lingo scripts for the Canopy movie is examined in detail in this section.

The Score
The Canopy movie makes limited use of the Score. There are essentially two markers around which the movie loops: the "Mov" marker (if the Canopy movie is playing a QuickTime video) or the "Im" marker (if the Canopy movie is playing a particle animation). The only other frame script is a hold script, which is run when the movie is first initialized. Once a message is received from the Book or the Radio; the playback head jumps to the Mov or Im markers. All other effects are handled by Lingo, rather than by moving the playback head.

The "Mov" loop
When a QuickTime movie is playing, the playback head loops around the Mov marker. There are only two sprites in the score: two QuickTime movies, which are alternately moved on- and off-stage by the Video Mixer object (described below). The frame script which runs the loop also sends a message to the Display object each time it's run (which should be roughly 10 times a second), so that the Display object can keep some real-time information synchronized.

The "Im" loop
When a particle animation is playing, the playback head loops around the Im marker. There are 12 sprites in the score, all attached to a dummy member and all with the same behavior. When the loop is running, the sprites attach themselves to bitmap cast members and are animated through Lingo. As with the Mov loop, the frame script sends a message to the Display object each time it's run, so that the Display object can sync up its real-time handlers.

Lingo Architecture: Abstract
The communication between the Lingo objects in the Canopy movie is shown in Figure 2, below.

Figure 2: Lingo Architecture for Canopy Movie

Messages originate outside the Canopy movie and are sent to the Mailbox object via the Network object. If the message originates from a Bed sensor, no media is changed, and so the Logic object is not engaged; instead, the Mailbox object sends the message directly to the Display object, which responds by changing the display. If the message originates from the Radio or Book movies, the Mailbox object sends a message to the Logic object, which performs the following actions:

It determines if new media needs to be displayed;

If so, it polls the Database object to find the proper media to display;

It directs the Display object to display the new media; and

It informs the Mailbox object of the change.

Finally, the Display object has a direct relationship with the Database object as well, since it occasionally needs to find the proper audio media to play during the particle animations. Because this does not require complex data mining, the Display object does not need to engage the Logic object. These objects are described in detail below.

The Internal Cast
The initialization script and the script that creates the Mailbox object are stored in the Internal cast.

Initialization
When the Canopy movie first starts up, it runs the following handler:

```
global gMailboxObject, gLogicObject, gDatabaseObject, gDisplayObject
on startMovie
```

```
voidVariables
startTimer
gDatabaseObject = new(script “Database Object Constructor”)
gLogicObject = new(script “Logic Object Constructor”)
gMailboxObject = new(script “Mailbox Object Constructor”)
gDisplayObject = new(script “Display Object Constructor”)
gLogicObject.initialValues()
gDisplayObject.initialValues()
gDatabaseObject.initialValues()
— 4/9/01: These are initialization values. The moment we receive data from the radio and the book, these will be overwritten.
gMailboxObject.receivedMessage(“BookCall”, 10) — This tells the movie to start on page x.
gMailboxObject.receivedMessage(“RadioCall”, 1) — This tells the movie that the initial mood is y.
end on voidVariables
gMailboxObject = VOID
gLogicObject = VOID
gDatabaseObject = VOID
gDisplayObject = VOID
end
```

The four major objects (Mailbox, Logic, Database, and Display) are created here. (The fifth object, gMServer, is a carry-over from the Network movie.) The voidVariables handler voids these objects, which prevents any old values from being maintained from an earlier session. Then, the objects are instantiated. The Logic, Database, and Display objects all have internal properties that can only be created once the object itself exists and is stored in a global variable, and so they are initialized in a 2-step process: the object is instantiated, and then its initialValues() handler is run, which creates its properties. The Mailbox object does not need this extra step, and so it is fully functional upon initialization.

Finally, two messages are sent to the Mailbox object to provide initial settings for mood and page. From this point, all other messages come from the Bed, Book, and Radio movies.

findHeaderNum(“H4”); The Mailbox object

```
— ============================================
— 4/8/01. Author: David Johnson
— This script establishes the Mailbox object.
— gMailboxObject has the following functions:
— a) a place to store all unattached global variables; and
— b) a routing station, interfacing to the Shockwave Multiuser server (gMServer). This is handled by
receivedMessage(newSubject,newContent), below.
— ============================================

property pPage, pMood, pMediaIndex
global gLogicObject, gDisplayObject
on new me
— Initialize the state variables.
— These are merely initial values; they’re almost immediately replaced.
pMediaIndex = [#Im, 1]
pPage = 1
pMood = 4
return me
end
— ============================================
— The receivedMessage() handler is the post office, which routes messages from the Shockwave MU Server. It calls the
createCompleteMessage() handler to complete the message, and then routes it to runAction().If the message was received from the Radio
movie, it converts the value, sent as a percentage, into the appropriate mood index between 1 - 9.
— ============================================
on receivedMessage me, newSubject, newContent
case (newSubject) of
— Book or Radio: create complete message and send it to the Logic object.
```

```
“BookCall”:
newList = me.createCompleteMessage(#book, newContent)
gLogicObject.runAction(#book, newList)
“RadioCall”:
newMood = me.convertContentToMood(newContent)
newList = me.createCompleteMessage(#radio, newMood)
gLogicObject.runAction(#radio, newList)
— Bed: send message to the Display object.
“BedCall”: gDisplayObject.runAction(#bed, newContent)
end case
end
on createCompleteMessage me, newSubject, newContent
case (newSubject) of
#book: — The external user has turned a page in the book.
oldMood = me.getMood()
return [newContent, oldMood]
#radio: — The external user has turned a knob on the radio.
oldPage = me.getPage()
return[oldPage, newContent]
end case
end
on convertContentToMood me, contentPercentage
minMood = 1
maxMood = 9
newMood = ( (float(contentPercentage)/100)*( maxMood - ( minMood - 1) ) )
if newMood < minMood then newMood = minMood
if newMood > maxMood then newMood = maxMood
return integer(newMood)
end
— ============================================
— The following handlers read/write the state variables.
— ============================================
on getPage me
return pPage
end
on setPage me, newPage
pPage = newPage
end
— ============================================
on getMood me
return pMood
end
on setMood me, newMood
pMood = newMood
end
— ============================================
on getIndex me
return pMediaIndex
end
on setIndex me, whichIndex
pMediaIndex = whichIndex
end
— ============================================
— cleanUp() is called when the movie stops. This object has nothing in particular to clean up.
— ============================================
on cleanUp me
nothing
```

end
As the comments indicate, gMailboxObject holds all unattached global variables (namely, the current page, mood, and media object being displayed), and it routes messages from the Server object to the other objects in the Canopy movie.

The property variables for page (pPage) and mood (pMood) are simply integers; these are set when the Logic object calls the accessor methods in the Mailbox object. The media index is stored as a linear list, in which the first value is a symbol (either #Im or #Mov, depending on whether particle animations or QuickTime movies are being displayed), and the second value is the specific item.

gMailboxObject has two types of methods. The second group are a series of accessor methods, allowing gLogicObject to read and write the values of the property variables; these should be straightforward. What's interesting, however, is the series of methods in the middle, which handle the routing of messages from the Server object. The crucial method is receivedMessage(), which receives messages from either the Book or Radio movies and routes them to the Logic object. As noted in the comments, the Logic object does not maintain any memory of the old mood or page, and so it's the Mailbox object's responsibility to send a "complete" message so that the Logic object has enough data to perform its functions. This means adding the mood value to a message received from the Book, or a page value to a message from the Radio.

The Book movie sends exact page numbers, and so there's no need to translate its messages. The Radio movie, on the other hand, converts its station numbers into a percentage value and then sends the percentage, and so the Mailbox object has to convert its messages into mood indices. The reason for this is to reduce implementation knowledge between the Radio and Canopy movies; there's no reason for the Canopy movie to know how many stations are on the radio. Because there is media that is specifically tied to pages of the book, however, the Canopy movie does need to know the precise page number being displayed.

Once the Mailbox object has compiled and translated the message, it sends it to the runAction() handler in the Logic object. From this point, the Mailbox object is entirely passive. Its only further interaction is to store values for mood, page, and media index.

Finally, all top-level objects have a cleanUp() method, which is called when the movie is stopped. The Mailbox object doesn't have anything in particular to clean up, so its handler currently does nothing. (The Display object is currently the object which uses this handler; the other objects don't yet make use of it.)

The Database Object Scripts Cast
The scripts that create the Database object are stored in the Database Object Scripts cast.

Overview
The Database object stores all of our media (images, QuickTime videos, and sound files) in a series of lists: the Image list (pImageMembers), the Movie list (pMovieMembers), and the Music list (pMusicMembers). Upon initialization, each of these lists performs a series of calls to an object constructor, which returns an object that points to a cast member. These cast members are stored in the "Movies," "Sounds," and "Final Bitmaps" casts.

Once the Database object has initialized the lists, which in turn have populated themselves with these objects, the Database object creates another list, the Book Index list (pBookIndexList). This list indexes all of the members in the Movie, Image, and Music lists with respect to the appropriate page of the Book movie. For example, if the Book is turned to page 1, the Book Index list can return a list of all the media that is appropriate to display for page 1 of the Book. Rather than storing these indices, we used to run the routines that mined the data members every time a page in the book was turned, but we changed the code to create the index list at startup so that the movie runs more quickly (with a slight cost in memory overhead).

This architecture is diagrammed below.

Creating the Database Object
The Database object is created when the Canopy movie is started. The object is created by the following script.

```
— ===========================================

— 4/8/01. Author: David Johnson
— This script establishes the Database object.
— gDatabaseObject has the following functions:
— a. It stores all media in a series of lists.
— b. It creates a series of index lists, which the Logic object uses to mine the database.
— ===========================================
property pImageMembers, pMusicMembers, pMovieMembers, pBookIndexList
— ===========================================
— The initialization scripts (new() and initialValues()) perform the following:
```

```
— a. new() creates lists of all media (#Im, #Mov, and #Song) and stores them in the property variables (e.g., all images are stored in pImageMembers, etc.) and an initial index value;
— b. initialValues() creates an index of the databases. Any time gLogicObject needs a piece of media, it queries the index. (This is to provide speedier response after initial loading.).
— ==========================================
on new me
— 11/27: pImageMembers, pMusicMembers, pTextMembers, and pMovieMembers are the objects that hold the “database.”
pImageMembers = new(script “Image List Object Constructor”)
pMusicMembers = new(script “Music List Object Constructor”)
pMovieMembers = new(script “Movie List Object Constructor”)
return me
end
on initialValues me
me.initializeIndexLists()
me.logBookIndices(pImageMembers, #Im)
me.logBookIndices(pMusicMembers, #Song)
me.logBookIndices(pMovieMembers, #Mov)
end
— ==========================================
— The index-creation scripts (initializeIndexLists() and logBookIndices()) perform the following:
— a. initializeIndexLists() creates the structure of pBookIndexList, which is the master list of indexes (that is, the list that relates all the media to each page of the book); and
— b. logBookIndices() populates pBookIndexList with values.
— These scripts are called upon initialization and then are never called again.
— ==========================================
on initializeIndexLists
global pBookIndexList
pBookIndexList = [:]
addProp pBookIndexList, #MotherPage, [:]
addProp pBookIndexList, #ChildPage, [:]
addProp pBookIndexList, #FatherPage, [:]
addProp pBookIndexList, #MarriagePage, [:]
addProp pBookIndexList, #BirthPage, [:]
addProp pBookIndexList, #ChangePage, [:]
addProp pBookIndexList, #AcceptancePage, [:]
addProp pBookIndexList, #PeacePage, [:]
addProp pBookIndexList, #DeathPage, [:]
addProp pBookIndexList, #BookClosed, [:]
repeat with counter = 1 to pBookIndexList.count()
addProp pBookIndexList[counter], #Im, []
addProp pBookIndexList[counter], #Song, []
addProp pBookIndexList[counter], #Mov, []
end repeat
end
on logBookIndices me, whichDatabase, whichLabel
— 4/9: this is based on the following page numbers:
— 1. Mother
— 2. Child
— 3. Father
— 4. Marriage
— 5. Birth
— 6. Change
— 7. Acceptance
— 8. Peace
— 9. Death
— 10. Book closed
mediaList = whichDatabase.getList()
```

```
repeat with counter = 1 to mediaList.count() — Go through each media object
myBitmask = mediaList[counter].getPageBitmask()
repeat with subCounter = 1 to myBitmask.length — Go through each bit and log the object each time there's a hit.
if value(myBitmask.char[subCounter]) = 1 then
add pBookIndexList[subCounter][whichLabel], counter
else
nothing
end if
end repeat
end repeat
end
— ============================================
```

The code above should be self-explanatory: the three media lists are created upon startup, and then the index list is created when the initialValues() handler is run. Each object in each of the media lists contains a pseudo-bitmask that identifies which pages of the book are appropriate for it. The logBookIndices() handler uses this bitmask to create the index list.

When a new piece of media is needed (either because a page in the Book was turned, or because the knob on the Radio was turned sufficiently), the Logic object calls the getBookMedia() handler, below, to provide a list of all the appropriate media for the current page of the Book. When the Display object needs data for a song title, it calls the following handler, getMediaObject(), to return the object associated with that song title.

```
— ============================================
— The database-mining scripts (getMediaObject() and getBookMedia()) perform the following:
— a. getMediaObject() receives an index and a media type, and returns the OBJECT associated with that index; and
— b. getBookMedia() receives a page number and a media type, and returns the list of all OBJECTS of that media type that are associated with that page.
— These scripts are called repeatedly by the Logic object.
— 4/15: NOTE: getMediaObject() is called by the Display object.
— ============================================
on getMediaObject me, mediaIndexList
mediaType = mediaIndexList[1]
mediaIndex = mediaIndexList[2]
case (mediaType) of
#Mov:
return pMovieMembers.getObject(mediaIndex)
#Song: — This is usually sent as a title, because it's linked to an #Im or a #Mov. Titles are strings; indices are integers.
case (mediaIndex).ilk of
#string:
objectIndex = pMusicMembers.titleToIndex(mediaIndex)
return pMusicMembers.getObject(objectIndex)
#integer:
return pMusicMembers.getObject(mediaIndex)
end case
#Im:
return pImageMembers.getObject(mediaIndex)
otherwise:
alert("getMediaObject: I don't recognize media type " & mediaType)
end case
end
on getBookMedia me, whichPage, whichMediaType
newList = []
— EXAMPLE: pBookIndexList[#MotherPage: [#Im: [1, 4], #Song: [3, 10], #Mov: [3, 10, 12]... ]
case (whichMediaType) of
#Im: myMediaListObject = pImageMembers
#Song: myMediaListObject = pMusicMembers
#Mov: myMediaListObject = pMovieMembers
end case
```

```
— Now, we send back the OBJECTS, not the indices.
if whichPage <> 0 then
repeat with counter = 1 to pBookIndexList[whichPage][whichMediaType].count()
myIndex = pBookIndexList[whichPage][whichMediaType][counter]
add newList, myMediaListObject.getObject(myIndex)
end repeat
else
newList = [0]
end if
return newList
end
— ==========================================
— cleanUp() is called when the movie stops. This object has nothing in particular to clean up.
— ==========================================
on cleanUp me
nothing
end
```

Finally, all top-level objects have a cleanUp() method, which is called when the movie is stopped. The Database object doesn't have anything in particular to clean up, so its handler currently does nothing.

Creating the Movie List

One database is pMovieMembers. This database is created by two object constructors. The first object constructor, the Movie List Object Constructor, creates one large linear list, which serves as the database for the QuickTime videos. Each entry in the linear list is an object, which is created by the second object constructor, the Movie Properties Object Constructor. The List constructor calls the Properties constructor once for each movie, passing in a series of values that define the properties of the movie, such as the mood it is meant to represent, the page in the Book that the movie applies to, the cast member, etc.

One of the advantages of using the List constructor is that all the properties for the movies are laid out in one script, which allows us to edit them in one central place. There are currently over 20 images, each with a unique series of properties, and so this centralization allows us to add or delete properties for all members quickly, as well as allowing us to edit the individual values for a member.

The Movie List Object Constructor

The script below shows the creation of the List object, along with a couple of exemplary objects. These objects are created by a call to the Properties constructor, and then the objects are stored in a linear list, pMovieGroupList.

pMovieMembers is created by the following script.

```
property pMovieGroupList
on new me
— 4/15/01: the movie cast members are all held in a linear list, pMovieGroupList.
— The attributes are as follows:
— pGroupLabel: the label, as a linear list
— pMemberList: a linear list including the cast members, as strings
— pLinkedMediaList: a linear list including lists of linked cast members (e.g., sounds), as strings. [0] means none.
— pPageBitmask: a bitmask that identifies which page(s) of the book can call this group. 1 = yes, 0 = no. IMPORTANT: this is represented as a STRING, so that we can identify a particular bit.
— pMood: integer: 1-9
— pCounterLimit: integer; number of times this is seen before it's locked out.
— pCounterTimeout: in minutes: amount of time this image is locked out before it's visible again.
— pPauseTime: linear list: where the movie left off the last time the viewer saw it. [memberNum, pauseTimeInTicks]
pMovieGroupList = []
— HERE IS AN EXAMPLE OF A LINKED MEDIA LIST:
— add pMovieGroupList, new(script "Movie Properties Object Constructor", [#Mov, 1], ["acceptance_efx"], [#Song:["seeingyou_echo", "darkcold", "sad_38"]], "111110111", 2, 3, 15, [1, 0])
— HERE IS AN EXAMPLE OF A MOVIE DIVIDED INTO A GROUP:
— add pMovieGroupList, new(script "Movie Properties Object Constructor", [#Mov, 7], ["Conformity1_1", "conformity1_2", "conformity1_3", "conformity1_4", "conformity1_5", "conformity1_6"], [#Song:["chet_baker", "esquivel"]], "000001000", 1, 3, 15, [1, 0])
add pMovieGroupList, new(script "Movie Properties Object Constructor", [#Mov, 1], ["mother_funeral"], [0], "1000000000", 2, 3, 15,
```

[1, 0])
add pMovieGroupList, new(script “Movie Properties Object Constructor”, [#Mov, 21], [“peace”], [0], “0000000110”, 8, 3, 15, [1, 0])

As demonstrated, each item in pMovieGroupList is created by a call to the “Movie Properties Object Constructor” script, along with a series of arguments. The object constructor returns an object, which is stored in this list. In the code snippet above, two entries are shown, for “mother_funeral.mov” and “peace.mov”; obviously, we have many more QuickTime movies than that, but these two entries are sufficient to explain how this object works.

The first thing to note is that the properties described in the comments, such as pMemberList and pPageBitmask, are NOT properties of the List constructor. These are properties of the individual objects (which is to say, they are property variables in the Properties constructor script).

Let’s go through the attributes one by one:

pGroupLabel: this is just a unique identifier for a particular movie. It is structured as a linear list; the first entry is a symbol, identifying the media type (always #Mov), and the second entry is an integer. For example, “mother_funeral” is the first entry; “peace,” the 21st.

pMemberList: this may seem odd. Why do we use a list of strings, rather than just a single string, to identify the QuickTime cast member? There are two reasons for this:

* The Image list has to use lists of cast members, and so it’s best if the structure of the Movie list is consistent; and

* There’s a historical reason as well. At one point, we would divide any QuickTime movie over 20MB into a series of smaller movies, and then play them in sequence, in order to reduce memory overhead. This lead to a series of complications in the Display object, which had to constantly keep track of the movie time and prepare to switch from one movie to another in order to present the illusion of a single movie. We finally scrapped this approach and kept our movies whole, but much of the code was built around deciphering this list architecture, and so we maintained it even though each list only contains a single string.

pLinkedMediaList: this contains pointers to any media (in particular, sounds) that may be played simultaneously with this media. There are (currently) no entries in the Movie list that have linked media, but all the entries in the Image list take advantage of this.

pPageBitmask: this is not a true bitmask, since Lingo does not provide bitwise operators. This is a string, structured to emulate a 10-bit bitmask, that identifies which page of the Book will call up this media (bit 10 = book closed). For example, “mother_funeral” can only be displayed for page 1 of the Book, and so the first “bit” is a 1 and the other 9 “bits” are zeros. “peace,” on the other hand, can be displayed for either the 8th or 9th pages of the Book.

pMood: when there is more than one movie that can be displayed for a given page of the Book (which is almost always true), we determine which movie to show on the basis of “mood” (which is essentially a function of the Radio station currently playing). The Logic object examines all of the media for the current page and finds the one that most closely fits the current mood in the installation; this is described in the discussion of the Logic object, below. For example, “mother_funeral” has a mood of 2, which means it’s among the darker, more depressing movies that fit page 1 of the Book. “peace” has a mood of 8, which means it’s among the happier, more lively movies that fit pages 8 and 9.

pCounterLimit and pCounterTimeout: in order to avoid having one piece of media dominate the installation, the Display object keeps track of the number of times a particular piece of media has been played. If “mother_funeral,” for example, is seen in its entirety more than three times in a 15-minute period, it’s locked out and cannot be displayed again for another 15 minutes, allowing the other media for page 1 to be displayed. NOTE: this is not a “hard” lockout (or else we would have nothing to display on page 4, since there’s only one movie for that page). If every movie for a page is locked out, the movie searches for the one that’s “least locked out.”

pPauseTime: if a movie is interrupted while it’s playing (which happens constantly), the Display object logs the point at which the movie stopped playing here. When the movie resumes, it plays from this point, not from the beginning. This is stored as a list, for the same historical reasons that the member name is stored in a list: if we divide a movie up into chunks, we need the index of the chunk, as well as its pause time. In practice, the first number is always 1, because each movie is an atomic unit.

There are two other handlers in the pMovieMembers object: getList(), and getObject().

```
— ===========================================

— 4/14: getList() and getObject() are the two default methods for each list object constructor.
on getList me
return pMovieGroupList
end
```

```
on getObject me, whichIndex
return pMovieGroupList[whichIndex]
end
— ==========================================
```
These should be self-explanatory.

WaterThe Movie Properties Object Constructor

Each entry in the Movie list is created by a call to the Movie Properties Object Constructor, as described above.

```
— ==========================================
— 4/15/01. Author: David Johnson
— This is a movie group database object. It stores the handlers for one movie group.
— ==========================================
property pGroupLabel, pMemberList, pLinkedMediaList, pPageBitmask, pMood, pCounterLimit, pCounterTimeout, pViewList,
pPauseTime
on new me, myGroupLabel, myMemberList, myLinkedMedia, myPageBitmask, myMood, myCounterLimit, myCounterTimeout,
myPauseTime
— pGroupLabel: the label, as a linear list
— pMemberList: a linear list including the cast members, as strings
— pLinkedMediaList: a linear list including lists of linked cast members (e.g., sounds), as strings. [0] means none.
— pPageBitmask: a bitmask that identifies which page(s) of the book can call this group. 1 = yes, 0 = no. IMPORTANT: this is represent-
ed as a STRING, so that we can identify a particular bit.
— pMood: integer; 1-9
— pCounterLimit: integer; number of times this is seen before it's locked out.
— pCounterTimeout: in minutes: amount of time this image is locked out before it's visible again.
— pPauseTime: linear list: where the movie left off the last time the viewer saw it [member, pauseTimeInTicks]
pGroupLabel = myGroupLabel — integer
pMemberList = myMemberList — linear list
me.verifyMembers(pMemberList, "Movies")
pPageBitmask = myPageBitmask — string (10 chars long)
me.verifyBitMask(pPageBitmask)
pLinkedMediaList = myLinkedMedia
repeat with n = 1 to pLinkedMediaList.count()
case (pLinkedMediaList[n]) of
0: nothing
otherwise:
mediaList = pLinkedMediaList[n]
mediaType = pLinkedMediaList.getPropAt(n)
case (mediaType) of
#Song: me.verifyMembers(mediaList, "Sounds")
otherwise: nothing
end case
end case
end repeat
pMood = myMood — integer (0-20)
pCounterLimit = myCounterLimit
pCounterTimeout = myCounterTimeout
pViewList = []
pPauseTime = myPauseTime
return me
end
on verifyMembers me, memberList, castName
repeat with n = 1 to memberList.count()
thisMember = memberList[n]
if member(thisMember, castName).number = -1 then
alert("Movie Properties Object Constructor: member " & thisMember & " doesn't exist.")
end if
end repeat
```

```
end
on verifyBitMask me, maskToCheck
numOfPages = 10
if maskToCheck.length <> numOfPages then
alert("Movie Properties Object Constructor: the bit mask for group " & pGroupLabel & " is the wrong length.")
end if
end
```

Much of this is self-explanatory: the object receives an argument and stores it as a property variable. The only interesting aspects are the "verify" handlers. These are here for authoring, rather than run-time, purposes: while editing the list above, it's disconcertingly easy to type in the wrong number for a bitmask, or to misspell the name of a movie, and so we have checks to make sure that this is caught before the movie is displayed publicly.

If this were all the object did, there would be no need to create an object at all; the Movie list itself is a sufficient data structure to store persistent variables. The reason we use an object-oriented approach is in order to bind the handlers below with the data.

```
— ============================================

— evaluateCounter() works with the Importance object to determine whether a media object has already been seen too often to be seen again.
— evaluateCounter() evaluates pViewList with respect to pCounterLimit. The logic is as follows:
— 1. Has the item been seen LESS than the max. number of times? If so, tell the caller (return 1).
— 2. If the item has been seen MORE than the max. number of times, tell the caller how many times it's been seen, and when the last time was.
— addSighting() populates pViewList. It adds an entry to pViewList when the END of an item is reached. It is called by the Display object.
— ============================================
on evaluateCounter me
thisMinute = (the timer)/3600
repeat with n = 1 to pViewList.count
if pViewList[n] <= thisMinute - pCounterTimeout then deleteAt(pViewList, n)
end repeat
if pViewList.count < pCounterLimit then
return 1
else
return [pViewList.count - pCounterLimit, pViewList[pViewList.count]]
end if
end
on addSighting me
add pViewList, (the timer)/3600
end
— ============================================
— Other handlers. ALL media objects have the following handlers.
— ============================================
on getMemberList me
return pMemberList
end
on getMood me
return pMood
end
on getPageBitmask me
return pPageBitmask
end
on getIndex me
return pGroupLabel
end
— ============================================
— getLinkedMedia() is specific to the #Mov and #Im objects.
```

```
— ============================================
on getLinkedMedia me, typeOfMedia
case (pLinkedMediaList[1]) of
0: return 0
otherwise: return pLinkedMediaList[typeOfMedia]
end case
end
— ============================================
— The pause time handlers are specific to the movie groups. If the movie is paused during playback, the Display member can log the pause time here and then resume here when the group is called again.
— ============================================
on getPauseTime me
return pPauseTime
end
on setPauseTime me, whichMemberNum, whichTime
pPauseTime = [whichMemberNum, whichTime]
end
```

Of these handlers, the most intriguing is evaluateCounter(). This handler works with the Logic object to determine whether or not this movie has been seen too frequently and should be locked out. When a movie is showing, the Display object notes if the end of the movie was seen. When the movie is changed, the Display object calls addSighting() if the end of the movie was seen, and addSighting stores the time at which this occurred. When evaluateCounter() is called, it deletes any entries from the view list that precede the number of minutes specified by pCounterTimeout, and then it decides whether the movie has been seen to the end more times than pCounterLimit allows. If not, it returns a 1 to the Logic object; if so, it returns a list to the Logic object, specifying how many times over the limit this movie has been seen, and when the last showing was. This is information that the Logic object uses to determine just how locked out a movie is.

The next group of handlers are just accessor methods. All media objects (#Im, #Mov, and #Song) have these.

Finally, getPauseTime() and setPauseTime() are used to allow a movie to display from where it left off, rather than from the beginning.

Creating the Image and Music Lists

pImageMembers and pMusicMembers are created in exactly the same fashion as pMovieMembers. An example of each is shown here, in order to describe properties that are unique to these media types.

First, the Image list:

```
property pImageGroupList
on new me
— 4/13/01: the bitmaps are grouped. The groupings are then held in a linear list, pImageGroupList.
— The attributes are as follows:
— pGroupLabel: the label, as a linear list (used as an index)
— pMemberList: a linear list including the cast members, as strings
— pLinkedMediaList: a linear list including lists of linked cast members (e.g., sounds), as strings. [0] means none.
— pPageBitmask: a bitmask that identifies which page(s) of the book can call this song. 1 = yes, 0 = no. IMPORTANT: this is represented as a STRING, so that we can identify a particular bit. NOTE: #10 means the book is closed.
— pMood: 1-9
— pColor: the color we use to tint the pictures.
— pCounterLimit: integer; number of times this is seen before it's locked out.
— pCounterTimeout: in minutes: amount of time this image is locked out before it's visible again.
pImageGroupList = []
— Angels
add pImageGroupList, new(script "Image Properties Object Constructor", [#Im, 1], ["5_2", "5_3", "6_1", "6_2", "7_1", "9_4", "10_4"], [#Song:["angels_9_15", "spkymumble_9_3", "churchbell_4_8"]], "0000000001", 1, rgb(255, 0, 0), 3, 15)
```

Most of this should look familiar. The Display object uses the image cast members in a single image group to generate a particle animation. These cast members are named by their row and column numbers in the curtain that hangs in the installation space (for example, the image in the 5th row, 2nd column is member "5_2").

There are a couple of differences from the Movie list. For one, the Image list refers to a series of linked #Song cast members, which play

during the particle animation. For another, there's a new attribute, pColor, which is used to tint the particle animations. The Image List Object Constructor has an accessor method, getColor(), which gives the Display object access to this property.

Finally, the Music list:

```
property pMusicList
on new me
— 11/15/00: the songs are all held in a linear list, pMusicList.
— The arguments that the Music Properties Object Constructor accepts are:
— pGroupLabel: the label, as an integer
— pMemberList: a linear list including the cast members, as strings. NOTE: even if there's only one song per group, we're still using lists, rather than single strings, to represent titles. This ensures consistency with the #Im and #Mov objects.
— pPageBitmask: a bitmask that identifies which page(s) of the book can call this group. 1 = yes, 0 = no. IMPORTANT: this is represented as a STRING, so that we can identify a particular bit.
— pMood: 1-5 = sad, 6-10 = turbulent, 11-15 = hopeful, 16-20 = joyful
— pLoopIn: in milliseconds
— pLoopOut: in milliseconds. (If this = 0, there's no loop.)
— pAbruptness: #smooth/#sudden. This indicates whether the song should fade in gradually, and whether it should match the mood of the currently-playing song.
— pPauseTime: where the song left off the last time the viewer heard it.
pMusicList = []
add pMusicList, new(script "Music Properties Object Constructor", [#Song, 1], ["dreamlife_c_10"], "0000000001", 1, 0, 0, #sudden, 0)
```

The only attributes that merit comment are the pLoopIn/pLoopOut and pAbruptness properties. These were designed to allow us to loop a sound cast member between two specific points, and to determine how smoothly or abruptly we fade a member in and out. As it happens, these attributes are not being used in the installation as of June 2001, but we've kept them intact in case we decide to use them after all.

The Logic Object Scripts Cast

The scripts that create the Logic object are stored in the Logic Object Scripts cast.

Overview

The Logic object is the central brains of the movie. It coordinates the entire routine of selecting and displaying media (which is to say, the entire function of the Canopy movie in general) by responding to messages from the Mailbox movie, running the routines that search through the databases to find the proper media, and telling the Display object to "set the stage" (that is, display a particular piece of media).

The Logic object contains two sub-object: the Mining object, which mines the databases, and the Importance object, which decides which media is "important" enough to display. This architecture is diagrammed below.

Creating the Logic Object

The Logic object is created when the Canopy movie is started. The object is created by the following script.

```
— ============================================

— This script establishes the Logic object.
— gLogicObject has the following functions:
— a) the media selection process (deciding which media to display);
— b) Coordinating values among the other top-level objects (for example, sending gMailboxObject the new media index).
— ============================================
— These are the objects that do the work. Because they both list gLogicObject as an ancestor, they can only be created after new() is run.
property pMiningObject, pImportanceObject
global gMailboxObject, gDatabaseObject, gDisplayObject
— ============================================
— As with several of the global objects, the Logic object is constructed in two calls: the new() call, which initializes the object, and the initialValues() call, which creates the other properties (objects).
— The initialization scripts (new() and initialValues()) perform the following:
— a. new() merely creates the Logic object;
— b. initialValues() creates the two subordinate objects: pMiningObject (which mines the databases) and pImportanceObject (which determines which of the potential results deserve to be displayed).
```

```
— ============================================
on new me
return me
end
on initialValues me
pMiningObject = new(script “Mining Object Constructor”)
pImportanceObject = new(script “Importance Object Constructor”)
end
```

This should be self-explanatory.

The major purpose of the Logic object is to mine the database in response to a change in the Book or Radio, in order to find the media that should be displayed. The process begins when runAction() is called by the Mailbox object.

```
— ============================================
— runAction() is the handler that responds to the SMU Server. It handles the top-level processes of selecting and displaying media.
— The parameters are structured as:
— #book, [newContent, oldMood]
— #radio, [newContent, oldMood]
— The logic works as follows:
— 1. Get the old media.
— 2. Call pImportanceObject to determine if a change needs to be made.
— 3. If so, call the Mining object to search for an object that meets the new criteria.
— 4. Call setStage() to pass the result to the Display object.
— 5. Set the appropriate global variables.
— ============================================
on runAction me, newSubject, newContentList
— Step 1/2: do we need to change the media?
oldObject = gDatabaseObject.getMediaObject(gMailboxObject.getIndex())
changeOrNot = pImportanceObject.evaluateChange(newSubject, newContentList, oldObject.getMood())
newMood = newContentList[2]
gMailboxObject.setMood(newMood) — We set this even if the media doesn't change.
case (changeOrNot) of
1: — (As of 4/12, we expect this answer always to be 1 if newSubject = #book, and occasionally to be 1 if newSubject = #radio.)
newPage = newContentList[1]
— Step 3: get the new media object from the Mining object.
newObject = pMiningObject.runSearch(newPage, newMood, oldObject)
— Step 4: set the stage.
me.setStage(newObject)
— Step 5: set the appropriate global variables.
gMailboxObject.setIndex(newObject.getIndex())
gMailboxObject.setPage(newPage)
otherwise:
nothing
end case
end
— ============================================
— The handlers below offer support services to the processes invoked by runAction().
— The interface to the Database object is:
— a. getBookMedia() returns a list of objects that fit the requested page of the book.
— The interface to the Display object is:
— b. setStage() tells the Display object to display a media object.
— ============================================
on getBookMedia me, whichPage, whichMediaType
return gDatabaseObject.getBookMedia(whichPage, whichMediaType)
end
on setStage me, whichObject
gDisplayObject.setStage(whichObject)
end
— ============================================
```

— cleanUp() is called when the movie stops. This object has nothing in particular to clean up.
— ==========
on cleanUp me
nothing
end

runAction() is the crucial handler. As the comments indicate, it coordinates everything: finding the new media, telling the Display handler to display it, and setting the global variables in the Mailbox object. In doing this, it calls some handlers that we've already discussed: gDatabaseObject.getMediaObject(), gMailboxObject.setMood(), gMailboxObject.setPage(), and gMailboxObject.setIndex(). In addition, it calls other sub-objects of the Logic object, pImportanceObject and pMiningObject (described below). Finally, it calls setStage(), which is relayed to the Display object and causes the new media to be displayed.

Another handler in the top level of the Logic object is getBookMedia(), which is called by the Mining object and which provides an interface to the Database object. Finally, all top-level objects have a cleanUp() method, which is called when the movie is stopped. The Logic object doesn't have anything in particular to clean up, so its handler currently does nothing.

The Mining Object
The Mining object is created by and called by the Logic object to mine the media databases. NOTE: in keeping with our top-level architecture, the Mining object does not communicate directly with the Database object. Instead, it sets the Logic object as its ancestor and then uses the getBookMedia() method of the Logic object to provide communication services. What this adds in redundancy is more than offset by the cleanliness of the architecture (that is, in the ease in debugging).

— ==========

— 4/15/01. Author: David Johnson
— This script establishes the Mining object.
— gLogicObject.pMiningObject has the following functions:
— a) it runs the search to find the proper media object to display.
— The ancestor property is set to gLogicObject so that this object has access to other objects (e.g., the Database object) through calling functions such as me.getBookMedia().
— ==========
property ancestor
on new me
global gLogicObject
me.ancestor = gLogicObject
return me
end
— ==========
— runSearch() is the function that finds the appropriate media object to display. The media item to display will either be a #Mov (QuickTime) or #Im (image/image sequence) object.
— runSearch() performs the following:
— 1. It polls the Database object twice (once for movies, once for images) and concatenates the results, creating an index of all movies and images that match the page of the book (pageObjectList);
— 2. It looks for the currently-playing object in the list, and deletes it from the list if it finds it. This ensures a visible change;
— 3. It call findBestMoodFit() to select the best fit from the list; and
— 4. It returns the best fit to the calling function.
— ==========
on runSearch me, whichPage, whichMood, oldObject
— Step 1: get the appropriate media for this page of the book.
pageObjectList = duplicate(me.getBookMedia(whichPage, #Mov)) — The duplicate command prevents distorting the original list, which is passed BY REFERENCE, not by value
imObjectList = me.getBookMedia(whichPage, #Im)
repeat with n = 1 to imObjectList.count()
add pageObjectList, imObjectList[n]
end repeat
if pageObjectList.count() > 1 then — If there's only one object, there's no need to run the search, and we can't deleteAt.
— Step 2: delete the current object, if it's in the list.
repeat with i = 1 to pageObjectList.count()

```
if pageObjectList[i] = oldObject then
pageObjectList.deleteAt(i)
exit repeat
end if
end repeat
— Step 3: find the best fit.
bestFitObjectIndex = me.findBestMoodFit(pageObjectList, whichMood)
else
bestFitObjectIndex = 1
end if
— Step 4: return the result.return pageObjectList[bestFitObjectIndex]
end
— ============================================
— findBestMoodFit() sorts the list of media objects and finds the best fit. It creates a property list, moodIndexList, to provide a sorted (by appropriateness of mood) array of indexes into the list of objects.
— findBestMoodFit() performs the following:
— 1. It receives the list of objects and the mood to fit, and then queries each object to get its mood;
— 2. It calls a global function, convertValToString(), to get around a quirk in Lingo that would otherwise prevent us from using Lingo's "sort" function;
— 3. It sorts the moodIndexList by appropriateness of mood (using the alphabetic properties, leaving the numeric values intact. This is what allows us to both sort and to maintain our indexes);
— 4. It runs through the list of objects one by one, in mood index order, and calls pImportanceObject to evaluate each object.
— 4a) If pImportanceObject returns an affirmative, findBestMoodFit() returns the index to the calling function.
— 4b) If pImportanceObject returns a fractional value, it means that we should continue looking for a better candidate. If we don't find one, we sort the list of fractional values and use the highest value.
— ============================================
on findBestMoodFit me, listOfObjects, whichMood
moodIndexList = [:]
bestMatch = 0.0
finalIndex = 0
repeat with counter = 1 to listOfObjects.count()
newMood = listOfObjects[counter].getMood()
moodDifference = (whichMood - newMood).abs
— The only way to sort the index list by mood AND preserve the index order is to make the sort parameter a property and the index order a value. Lingo's "sort" feature works numerically for linear lists, and alphabetically BY PROPERTY for prop lists.
convertValToString() allows us to create the alphabetic properties correctly.
newProp = convertValToString(moodDifference)
addProp moodIndexList, newProp, counter
end repeat
sort moodIndexList
repeat with counter = 1 to moodIndexList.count()
myObject = listOfObjects[moodIndexList[counter]]
myPriority = me.pImportanceObject.evaluateImportance(myObject, moodDifference)
case (myPriority) of
1:
finalIndex = moodIndexList[counter]
return finalIndex
otherwise:
if myPriority > bestMatch then
finalIndex = moodIndexList[counter]
bestMatch = myPriority
else
nothing
end if
end case
end repeat
return finalIndex
```

end

runSearch() is the critical handler in this object, just as runAction() is in the Logic object. As the comments explain, runSearch() creates a list, pageObjectList, from a combination of all the #Mov and #Im media that fits the current page of the book. It calls findBestMoodFit() and passes it the list and the appropriate mood. findBestMoodFit() then indexes the list by mood, using the services of a global handler (that is, a handler in a movie script, rather than a parent or frame script) called convertValToString().

on convertValToString newVal
if newVal < 10 then
return "0" & string(newVal) — Obviously, we're only expecting 2-digit values; if we ever need to use this for 3-digit values, this will need to be expanded.
else
return string(newVal)
end if
end

convertValToString() allows us to sort a list by mood and maintain our original set of indices, even if the difference between the object's mood and the appropriate mood is 10 or greater. (As the current mood tops out at 9, this seems like an unnecessary step, but this degree of robustness allows us to change the mood granularity merely by changing the value in gMailboxObject.convertContentToMood(), without having to track down the effects of the change throughout the movie.)

Once findBestMoodFit() has sorted the list, it runs through the list from best fit to worst fit, passing each object to the Importance object for evaluation. This evaluation is described in the discussion of the Importance object, below; for now, it's sufficient to note that a return value of 1 means that the Importance object approves the showing of this media item. Once findBestMoodFit() receives a value of 1, it passes the index back to runSearch() and its job is over. If none of the media receives a 1 from the Importance object, however, findBestMoodFit() keeps a running tab of the best fit so far and returns it at the end of the search. Finally, runSearch() returns the best object to the Logic object, which calls the Display object to display it.

findHeaderNum("H4"); The Importance Object

The Importance object has two points of entry: it works with the top-level Logic object to decide whether or not there should be a change in media, and it works with the Mining object to determine if a piece of media "deserves" to be seen.

— ==========
— 4/15/01. Author: David Johnson
— This script establishes the Importance object.
— gLogicObject.pImportanceObject has the following functions:
— a. It tells the Logic object whether or not a change in the external environment (such as a book or radio change) requires a change in the internal environment;
— b. It tells the Mining object whether a particular piece of media deserves to be seen again.
— ==========
— pMoodLatitude describes how much a media object can differ from the requested mood and still be an appropriate fit.
property pMoodLatitude
on new me, currentMediaIndex
pMoodLatitude = 2 — on a scale of 1 - 9
return me
end
— ==========
— evaluateChange() is the function that tells the Logic object whether or not we should change the internal environment, in response to a call from the external environment.
— The logic is structured as follows:
— a. If the change request comes from the book, change the media.
— b. If the change request comes from the radio, evaluate the change:
— b1) If the change is within the mood latitude, don't change the media (since people will probably flip through the dial faster than through the book, and we don't want to break the Canopy movie by changing 20X a second).
— b2) If the change exceeds the mood latitude, change the media.
— ==========
on evaluateChange me, newSubject, newContentList, currentMood
case (newSubject) of
#book: return 1
#radio:
if newContentList[1] = 10 then

```
return 1
else
newMood = newContentList[2]
if (newMood - currentMood).abs >= pMoodLatitude then
return 1
else
return 0
end if
end if
end case
end
— ============================================
— evaluateImportance() receives a media object and tells the calling function whether or not this media was seen too recently, or too often, to be displayed again.
— The logic is as follows:
— 1. Has the item been seen LESS than the max. number of times? If so, show it (that is, return an integer value of 1). (We don't care how recently the viewer saw it if it's under the limit.)
— 2. If the item has been seen MORE than the max. number of times, return a fractional value, to describe the suitability of the object (higher numbers = more suitable: 0.9 deserves to be seen more than 0.3). The calling function can then decide whether the object should be seen.
— The logic that calculates the floating-point value is as follows:
— a. Mood appropriateness is weighted at 40% (lower mood difference = higher rating);
— b. The number of times the media has been seen is weighted at 30% (more times = lower rating);
— c. How recently the media has been seen is weighted at 30% (more recently = lower rating).
— ============================================
on evaluateImportance me, newObject, moodDifference
viewingCounter = newObject.evaluateCounter() — evaluateCounter() either returns a 1, meaning that this object has been seen fewer than the max number of times, or a linear list in the form [number of viewings over max (NOTE: this may be zero), time in minutes of last viewing].
case (viewingCounter) of
1: return 1
otherwise:
myFinalValue = 0.0
currentMin = (the timer)/3600
moodWeight = 0.4
numWeight = 0.3
viewWeight = 0.3
maxMoodValue = 1.0 — Since moodValue is fractional, the upper limit = 1.0.
maxNumViewings = 10 — We assign a max of 10 viewings over the limit; if the media has been seen 10+ times more than it should have been, the value for numOfViewings is zero.
maxNumMinutes = 60 — We assign a max of 60 minutes; if the media hasn't been seen for 60 minutes, the viewers have forgotten it (or there are new viewers) and the value for lastView is maximum.
moodValue = maxMoodValue - float(moodDifference)/pMoodLatitude
numOfViewings = float(maxNumViewings - viewingCounter[1])/maxNumViewings
if numOfViewings < 0 then numOfViewings = 0
lastView = float(currentMin - viewingCounter[2])/maxNumMinutes
if lastView > 1 then lastView = 1
myFinalValue = (moodValue*moodWeight) + (numOfViewings*numWeight) + (lastView*viewWeight)
return myFinalValue
end case
end
```

As described above and in the code comments, there are two points of entry into the Importance object. The Logic object calls evaluateChange(), and the Mining object calls evaluateImportance(). Both of these handlers rely on the "mood latitude." This magic number is the permissible deviation from the current mood. For example, if the current mood is 4 (moderate, tending towards dark and grim) and a piece of media with a value of 9 (extremely upbeat and happy) is being evaluated, it would fail because it's outside the mood latitude, which is a value of +/- 2 (as of June 2001; it's worth noting that we've tweaked this value repeatedly). Only media with moods

between 2 and 6 would fit.

evaluateChange() is quite straightforward. If somebody turns a page on the Book, change the media. If somebody changes the Radio dial, change the media if the new mood is outside the mood latitude. Even if the current media item might work on the new page (as "peace.mov" works when somebody changes from page 8 to page 9), the Mining object will ensure that a change takes place if at all possible.

evaluateImportance() is more complicated. It receives an object and is asked whether or not to display it. It queries the object's evaluateCounter() handler, which we examined earlier. If the object returns a value of 1, the Importance object simply returns it. However, if the media has been seen too many times, evaluateImportance() creates a weighted index for it on the basis of how often it's been seen, how recently it's been seen, and how closely it fits the requested mood. (Note the values for maxNumViewings and maxNumMinutes; these provide limits against which the handler can make the evaluation.) Once evaluateImportance() creates the fractional value, it returns it to the Mining object, which uses the value to find a best-fit media item in the event that none of the appropriate items for the page receives a value of 1. This is what we meant when we said that the object's viewing counter wasn't a "hard" limit; if everything is unavailable to be seen, we use this method to find the "least locked out" media.

The Display Object Scripts Cast
The scripts that create the Display object are stored in the Display Object Scripts cast.

Overview
The Display object controls the display of our three media types: QuickTime movies, particle system animations, and sounds. It has a series of sub-objects that handle each of these media types. In addition, the higher-level QuickTime ("Video Mixer object") and sound ("Audio Mixer object") objects instantiate and manage lower-level objects, which handle the display of media. The Particle object, on the other hand, is assisted by a series of behaviors, which are attached to sprites in the Score.

In addition, the Display object describes how the Canopy movie responds to pressure on the Bed sensors. (In this aspect, the Display object is the only top-level object that deals with the Bed movie; even the Logic object doesn't participate in this operation, because the Bed doesn't require any change in media.) The Display object has a lower-level object to deal with this function, as well. As with the Audio and Video mixers, this object (the "Bed Sensor object") manages a series of lower-level objects, each of which is responsible for a single Bed sensor. Unlike the Audio and Video mixers, however, these lower-level objects aren't instantiated from a single constructor script, but are instead initialized from a series of scripts.

This architecture is diagrammed below.

Creating the Display Object
The Display object is created when the Canopy movie is started. The object is created by the following script.

```
— ============================================
— 4/8/01. Author: David Johnson
— This script establishes the Display object.
— gDisplayObject has the following functions:
— a) Determine how the user sees and hears, in response to both the Logic object and the Bed movie (as routed by the Mailbox object).
— ============================================
property pAudioMixerObject, pVideoMixerObject, pParticleObject — Output objects: what the user sees and hears.
property pBedSensorObject — Input object: response to the Bed movie.
property pMediaObject, pMaxParticleFPS, pCurrentTempo — Variables
global gMailboxObject, gDatabaseObject
— ============================================
— As with several of the global objects, the Display object is constructed in two calls: the new() call, which initializes the object, and the initialValues() call, which creates the other properties (objects).
— The initialization scripts (new() and initialValues()) perform the following:
— a. new() merely creates the Display object;
— b. initialValues() creates the subordinate objects: pAudioMixerObject, pVideoMixerObject, pParticleObject, and pBedSensorObject.
— ============================================
on new me
return me
end
on initialValues me
pAudioMixerObject = new(script "Audio Mixer Object Constructor")
```

```
pVideoMixerObject = new(script “Video Mixer Object Constructor”)
pParticleObject = new(script “Particle Object Constructor”)
pBedSensorObject = new(script “Sensor Object Constructor”)
pMaxParticleFPS = 20
pCurrentTempo = pMaxParticleFPS
end
— ===========================================
— setStage() is the handler that responds to input from the Logic object (that is, from the book and radio).
— setStage() receives a media object and displays it on the stage.
— ===========================================
on setStage me, whichObject
pMediaObject = whichObject
pLinkedMediaList = pMediaObject.getLinkedMedia(#Song)
case (pMediaObject.getIndex()[1]) of
#Mov:
go to frame “Mov”
pParticleObject.cleanUp()
me.switchSensorMedia(#Mov)
pVideoMixerObject.setStage(pMediaObject)
pAudioMixerObject.setStage(pLinkedMediaList, #Mov)
#Im:
pParticleObject.cleanUp()
pVideoMixerObject.cleanUp()
me.switchSensorMedia(#Im)
pParticleObject.setStage(pMediaObject)
pAudioMixerObject.setStage(pLinkedMediaList, #Im)
go to frame “Im”
puppetTempo pMaxParticleFPS
end case
end
— ===========================================
— cleanUp() is called when the movie stops. The Particle object generates cast members that need to be erased.
— ===========================================
on cleanUp me
pParticleObject.cleanUp()
end
— ********************************************************************************************
—Everything from this point on determines how the Display object responds to input from the Bed sensors.
—********************************************************************************************
— BED SENSOR HANDLERS
— ===========================================
— The following handlers determine how the Bed sensor object responds to input from the Bed sensors.
— runAction() is the handler that responds to input from the bed. This is in direct response to the Mailbox object.
— The arguments for runAction() are:
— a. newSubject (which should always be #bed);
— b. newContent, in the form of a linear list. The first value is the stream that the input came from (on a scale of 0 to 5), and the second value is the “voltage” reading from that stream (on a scale from 1 to 100). For example, a list of [1, 20] means that stream 1 is sending a voltage of 20%.
— switchSensorMedia() tells the sensor objects whether a #Mov or #Im object is being displayed. The sensors may act differently as a result. The handler is called by setStage(), above.
— ===========================================
on runAction me, newSubject, newContent
case (newSubject) of
#bed:
pBedSensorObject.runAction(newSubject, newContent)
otherwise: nothing — 4/8: this should only be from the bed.
end case
```

```
end
on switchSensorMedia me, mediaType
pBedSensorObject.switchSensorMedia(mediaType)
end
-- ************************************************************************************
-- AUDIO/VIDEO MIXER HANDLERS
-- ==========================================
-- playAlternateTrack() switches from one linked sound track to another. Depending on the type of media playing, it can be sent to either the video or audio mixer.
-- changeVolume() reduces the overall volume. It is called from one of the sensors. Depending on the type of media playing, it can be sent to either the video or audio mixer.
-- ==========================================
on playAlternateTrack me, percentValue
case (pMediaObject.getIndex()[1]) of
#Mov: pVideoMixerObject.playAlternateTrack(percentValue)
#Im: pAudioMixerObject.playAlternateTrack(percentValue)
end case
end
-- ==========================================
on changeVolume me, percentRate
case (pMediaObject.getIndex()[1]) of
#Mov: pVideoMixerObject.changeVolume(percentRate)
#Im: pAudioMixerObject.changeVolume(percentRate)
end case
end
-- ==========================================
-- ************************************************************************************
-- AUDIO MIXER HANDLERS
-- ==========================================
-- getMediaObject() is called by the Audio Mixer Object. It receives its media as titles, rather than objects (because of the linked media list), and so it needs to query the database to get the proper attributes for its objects.
-- 4/21: we considered having the song titles replaced by their respective objects in the Database Object at start time, so that this step could be eliminated, but decided that it would introduce more delay at this point. Using titles means that the Audio Mixer Object can start playing a song immediately, without having to query the database or the object.
-- ==========================================
on getMediaObject me, mediaIndexList
gDatabaseObject.getMediaObject(mediaIndexList)
end
-- ==========================================
-- ************************************************************************************
-- PARTICLE SYSTEM HANDLERS
-- ==========================================
-- getSpriteVals() is called by the particle sprites, and allows them to load their properties from the Particle object.
-- ==========================================
on getSpriteVals me
spriteVals = me.pParticleObject.getSpriteVals()
return spriteVals
end
-- ==========================================
-- These handlers affect the display of the Particle system.
-- changeParticleRate() speeds or slows the Particle movies. It is called from one of the sensors.
-- changeParticleColor() tints the particles. It is called from one of the sensors.
-- setOpacity() changes the opacity of the particles. It is called from one of the sensors.
-- setOriginPoint() moves the particle system around the stage. It is called from one of the sensors.
-- getCurrentTempo() tells a frame script what the tempo should be, so that it can puppet the tempo.
-- ==========================================
on changeParticleRate me, percentRate
```

```
pCurrentTempo = pMaxParticleFPS - ( (percentRate*pMaxParticleFPS)/100 )
puppetTempo pCurrentTempo
end
— ==========================================
on changeParticleColor me, percentRate
pParticleObject.changeParticleColor(percentRate)
end
— ==========================================
on setOpacity me, percentRate
pParticleObject.setOpacity(percentRate)
end
— ==========================================
on setOriginPoint me, percentRate
pParticleObject.setOriginPoint(percentRate)
end
— ==========================================
on getCurrentTempo me
return pCurrentTempo
end
— ********************************************************************************
— VIDEO MIXER HANDLERS
— ==========================================
— recdFrameEvent() is called by the frame script that loops the QT movie sprites.
— ==========================================
on recdFrameEvent me
pBedSensorObject.recdFrameEvent()case (pMediaObject.getIndex()[1]) of
#Mov: pVideoMixerObject.recdFrameEvent()
otherwise: nothing
end case
end
— ==========================================
— changeMovieRate() is called by the Bed scripts, and changes the rate of the QT movie.
— ==========================================
on changeMovieRate me, whichDirection, whichRate
pVideoMixerObject.changeMovieRate(whichDirection, whichRate)
end
— ==========================================
— loopMovie() is called by the Bed scripts.
— ==========================================
on loopMovie me, whichAction
pVideoMixerObject.loopMovie(whichAction)
end
```

A couple of things are immediately apparent. First is the sheer number of handlers. Because all communication from the Bed sensors is routed through the Display object, there are many small handlers which exist mostly to pass information from the Bed sensor objects to the appropriate display systems (audio, video, and particle). Second is that the cleanUp() handler, which is dormant in the other top-level objects, has a function here, which will become clear when we examine the Particle object.

The initialization scripts (new() and initialValues()) are typically straightforward. The heavy lifting is done by setStage(), which is central to this object in the way that runAction() is central to the Logic object. SetStage() receives a media object and displays it by logging any linked media, calling the appropriate handlers in the Audio Mixer, Video Mixer, and Particle objects, and moving the playback head to the appropriate marker in the Score.

As noted in the comments, all the other handlers determine how the Display object reacts to input from the Bed sensors:

* Bed sensor object: runAction() and switchSensorMedia() call the Bed sensor object; runAction() relays messages from the Bed movie, while switchSensorMedia() tells the sensor objects whether they're affecting QuickTime videos or particle animations.

* Audio AND Video Mixer objects: playAlternateTrack() and changeVolume() call the Video Mixer object when a QuickTime video is playing, and call the Audio Mixer object when a particle animation is playing. Their purpose should be obvious; their implementation is discussed below.

* Audio Mixer object only: getMediaObject() continues our architectural philosophy of not permitting a lower-level object directly to call a top-level object. The Audio Mixer object is told to play a sound member, which it gets from setStage() in the form of pLinkedMediaList. It needs information from the object associated with that member, and so it routes its request to the Database object through the top-level Display object. The reason that the linked media list is comprised of titles, rather than objects, is to reduce memory overhead and to make creating pImageMembers more straightforward.

* Particle object: as noted above, the settings in the Particle object are transferred to the image sprites when their behaviors call the object; getSpriteVals() is the handler that the behaviors call. changeParticleRate() is handled by the Display object itself, since we chose to limit control of the score to the high-level objects only. The other handlers, such as changeParticleColor() and setOpacity(), are handled by the Particle system.

* Video Mixer object only: changeMovieRate() and loopMovie() are called by the sensors. recdFrameEvent() is a real-time call: as the play-back head loops through the Score, it sends calls to the Display object to keep internal events synchronized. (NOTE: the same thing could be accomplished by an onIdle() handler, but we wanted to avoid the overhead that that handler can produce.)

The Video Mixer and Video Channel Objects

One of the lower-level display objects is the Video Mixer. As with the databases discussed previously, this function is created by two object constructors. The first object constructor, the Video Mixer Object Constructor, handles the higher-level functions of displaying video and controls a series of video "channels." Each channel is managed by its own object, which is created by the second object constructor, the Video Channel Object Constructor. The Mixer constructor calls the Channel constructor once for each channel, passing in some values that define some properties of the channel, such as the sprite channel in the score over which the Channel object has control.

It's worth noting that the Mixer object deals with the media objects AS OBJECTS, parceling out information to the Channel objects as needed. A Channel object knows the cast member that it's playing, but it does not communicate with the object associated with that cast member. This aids debugging by preventing unexpected interactions to occur; by limiting the number of entry points into the media object, it's unlikely that a race condition will occur.

The Video Mixer Object Constructor

The script below shows the creation of the Video Mixer object. The need for a video mixer should be familiar to Lingo programmers working with QuickTime media: a common technique for avoiding the "white flash" that occurs when a QuickTime movie is started up is to start playing it off-stage, and then move the movie onto the stage once it's playing. Because we need to change seamlessly from one movie to the next, we are constantly moving QuickTime sprites on and off the stage.

Because our QuickTime movies play at full-frame, half-rate (15 fps), there's no way to play them as sprites and maintain our frame rate. Therefore, the "Direct to Stage" property is set to TRUE on all QuickTime cast members. In addition, the "Streaming" property is set to FALSE, in case we want access to the cue points in a member (currently, we are not using this feature).

```
— ============================================
— 4/22/01. Author: David Johnson
— This script establishes the Video Mixer object.
— gDisplayObject.pVideoMixerObject has the following functions:
— a. Coordinate and control the Quick Time movies.
— ============================================
property pVideoChannelList — Static variables: these are set at initialization.
property pFirstMovieSprite, pTotalMovieSprites — Constants
property pMediaObjectList, pMovieTitleList, pMovieTitleIndex, pCurrentVideoChannel — State variables: these change continually.
(Technically, pMediaObjectlist is set at initialization, but its contents are continually changing.)
global gDisplayObject
— ============================================
— new() has the following functions:
— a. Initialize any constants or static variables.
— ============================================
on new me
pFirstMovieSprite = 4
```

```
pTotalMovieSprites = 2
pVideoChannelList = []
pMediaObjectList = []
repeat with n = 1 to pTotalMovieSprites
mySprite = pFirstMovieSprite + (n - 1)
add pVideoChannelList, new(script “Video Channel Object Constructor”, mySprite, n)
sprite(mySprite).puppet = 1
add pMediaObjectList, 0
end repeat
return me
end
— ==========================================
— setStage() tells the video objects to begin and quit playing QT files.
— setStage() performs the following:
— a. Finds the title(s) for the member(s) to play;
— b. Finds the starting time (for example, if that movie was paused, we pick up at the pause point)
— c. Kills any objects that are currently playing;
— d. Finds an available video object;
— e. Sends the title to the object.
— clearStage() is called when gDisplayObject wants to play a different media object. When the movie is first started up, there are no
vales, so clearStage() has to protect against VOID values.
— clearStage() performs the following:
— a. Kills any objects that are currently playing;
— b. Tells the currently-playing object its pause time.
— ==========================================
on setStage me, newMediaObject
pMovieTitleList = newMediaObject.getMemberList()
startingTime = newMediaObject.getPauseTime()
pMovieTitleIndex = startingTime[1] — In most cases, this value will be 1, but some media objects may refer to multiple QT files.
pauseTime = startingTime[2]
pCurrentVideoChannel = findAvailableChannel()
pVideoChannelList[pCurrentVideoChannel].setStage(pMovieTitleList[pMovieTitleIndex], pauseTime)
pMediaObjectList[pCurrentVideoChannel] = newMediaObject
repeat with n = 1 to pVideoChannelList.count
if n <> pCurrentVideoChannel then
me.clearStage(n)
end if
end repeat
end
on clearStage me, whichChannel
if (pCurrentVideoChannel <> VOID) then
timeAndDurList = pVideoChannelList[whichChannel].clearStage()
case (pMediaObjectList[whichChannel]) of
0:
nothing
otherwise:
pMediaObjectList[whichChannel].setPauseTime(pMovieTitleIndex, timeAndDurList[1])
if timeAndDurList[2] = 1 then
pMediaObjectList[whichChannel].addSighting()
else
nothing
end if
end case
else
nothing
end if
end
```

```
— ============================================
— findAvailableChannel() queries the video objects, looking for an available one.
— ============================================
on findAvailableChannel me
repeat with n = 1 to pVideoChannelList.count
if pVideoChannelList[n].getStatus() = #Available then return n
end repeat
return 0
end
— ============================================
— Housekeeping. This prevents unused cast members from proliferating.
— ============================================
on cleanUp me
repeat with n = 1 to pVideoChannelList.count
me.clearStage(n)
end repeat
end
```

The Video Mixer object instantiate as many Video Channel objects as there are QuickTime sprites in the Score (in this case, 2), and stores them in pVideoChannelList. When each Video Channel object is instantiated, it sets a property variable that indicates its status (whether or not it's playing), among other things; this value is queried by findAvailableChannel().

The major work is done by setStage() and clearStage(). setStage() receives a new media object, finds an available channel (that is, a channel controlling a QuickTime sprite that's not currently playing), and then calls the setStage() handler in that Channel object. It then loops through the remaining Video Channel objects, including the one that was previously playing a movie, and calls clearStage(), which in turn calls their clearStage() handlers. The Video Channel objects that are being killed return information about any QuickTime movies they might have been playing, such as the point in the movie at which they stopped (their pause time) and whether or not the viewers saw the end of the movie. The Video Mixer object calls the setPauseTime() and addSighting() handlers in the media objects, if appropriate.

cleanUp() is called when the Particle object is started, to clear QuickTime sprites from the stage.

The remaining handlers for the Video Mixer object pass through calls from the Bed sensor objects to the currently-playing channel.

```
— ============================================
— changeMovieRate() is called by the Bed scripts, and changes the rate of the QT movie.
— ============================================
on changeMovieRate me, whichDirection, whichRate
pVideoChannelList[pCurrentVideoChannel].changeMovieRate(whichDirection, whichRate)
end
— ============================================
— loopMovie() is called by the Bed scripts.
— ============================================
on loopMovie me, whichAction
pVideoChannelList[pCurrentVideoChannel].loopMovie(whichAction)
end
— ============================================
— playAlternateTrack() is called by the Bed scripts.
— ============================================
on playAlternateTrack me, percentValue
pVideoChannelList[pCurrentVideoChannel].playAlternateTrack(percentValue)
end
— ============================================
— changeVolume() is called by the Bed scripts.
— ============================================
on changeVolume me, percentRate
pVideoChannelList[pCurrentVideoChannel].changeVolume(percentRate)
end
— ============================================
```

— recdFrameEvent() is called by the frame script that loops the QT movie sprites.
— ==========
on recdFrameEvent me
pVideoChannelList[pCurrentVideoChannel].recdFrameEvent()
end
These should be self-explanatory.
findHeaderNum("H4"); The Video Channel Object Constructor
Each QuickTime sprite in the Score is controlled by a Video Channel object, as described above.
— ==========
— 4/22/01. Author: David Johnson
— This script establishes a single video sprite.
— gDisplayObject.pVideoMixerObject has the following functions:
— a. Handle the media and placement of the Quick Time movies.
— b. Handle effects that are applied to Quick Time movies (e.g., masking, movement, cue point jumping, etc.)
— ==========
property pSprite, pIndex — Static variables: these are unique to each sprite and are set at initialization.
property pStageSize, pStageArea, pOtherArea, pWidth, pHeight — Constants: these are the same for every track.
property pAvailable, pMovieTitle, pRate, pCurrentRect, pQTAudioTrack, pStartTime, pOldPauseTime, pLoopState, pLoopLength, pLoopTime — State variables: this is unique to each track and changes continually.
— ==========
— The initialization scripts create the object and load on-screen and off-screen areas into the property variables.
— new() has the following functions:
— a. Call the findArea() handlers to determine the on- and off-screen areas.
— b. Initialize any constants or static variables.
— When the system changes from one video to the next, the only way to make the transition seamless (assuming Direct To Stage) is to move the currently-playing video completely off the stage; every other transition causes an unsightly jump. The findArea() handlers set the coordinates for this.
— findViewableArea() has the following functions:
— a. Find the area for a currently-playing movie to display.
— findOtherArea() has the following functions:
— a. Find an area where the movie will be completely hidden.
— ==========
on new me, mySprite, myIndex
— Static variables.
pSprite = mySprite
pIndex = myIndex
— Constants.
pStageSize = [640, 480]
pStageArea = findViewableArea(me, pStageSize)
pOtherArea = findOtherArea(me, pStageSize, pStageArea)
pWidth = 640
pHeight = 480
— State variables.
pAvailable = #Available — This means that the Video Mixer can send me a QT file to play. The states are: #Available, #Paused, and #Playing.
pQTAudioTrack = 1 — 5/18: we default to channel 1 if a MOV is playing.
pStartTime = the timer — 5/22: this is when a movie begins playing
pLoopState = 0 — 5/24: if this is set, the movie loops when me.recdFrameEvent() is called.
pLoopLength = 180 — 5/24: in ticks
return me
end
— ==========
on findViewableArea me, stageSize
—3/12: stageSize(1) is the width of the stage (e.g., 640); stageSize(2) is the height (e.g., 480). For today, I'm assuming that the top left pixel is (0, 0); if it's not, this is the handler to modify.
return rect(0, 0, stageSize[1], stageSize[2])
end

```
— ============================================
on findOtherArea me, stageSize, stageArea
safetyOffset = 10 — 3/12: this is to be absolutely sure that no edge pixels intersect
point1 = stageArea[1] - (stageSize[1] + safetyOffset) — 3/12: left side
point2 = stageArea[2] - (stageSize[2] + safetyOffset) — 3/12: top
point3 = stageArea[3] - (stageSize[1] + safetyOffset) — 3/12: right side
point4 = stageArea[4] - (stageSize[2] + safetyOffset) — 3/12: bottom
otherArea = rect(point1, point2, point3, point4)
case (intersect(otherArea, stageArea) = rect(0, 0, 0, 0)) of
1: return otherArea
otherwise: alert (“FindOtherArea: the stage area overlaps the off-stage area.”)
end case
end
— ============================================
— setStage() and clearStage() handle the playing of QT files.
— Starting a QT file is a 2-stage process:
— 1. Start playing the file; and
— 2. Move the file on-stage.
— These events should NOT happen simultaneously. Therefore, the setStage() handler forces the playback head to the next frame before moving the movie on-stage.
— Stopping a movie is handled similarly: the movie is moved off-stage first, and then stopped.
— setStage() performs the following:
— a. Begins playing the member;
— b. Sets all appropriate state variables;
— c. Sets a flag, pStartMe, which tells recdFrameEvent(), below, to move the movie to the stage on the next frame event.
— clearStage() performs the following:
— a. Moves off the stage;
— b. Sets variables that it returns to the calling function, pauseTime and numViewings; and
— b. Stops playing, sets pAvailable to #Available, and deletes the member reference.
— ============================================
on setStage me, memberName, startingTime
— 1. Start playback
sprite(pSprite).member = member(memberName)
— Start playing and play only the appropriate audio track.
sprite(pSprite).movieRate = 1
— Set variables
pAvailable = #Playing
go to the frame + 1
me.moveMovie(#OnStage)
sprite(pSprite).movieTime = startingTime
pStartTime = the timer
— State variables.
pMovieTitle = sprite(pSprite).member.name
pDuration = sprite(pSprite).member.duration
pOldPauseTime = startingTime
me.playAlternateTrack(0)
end
— ============================================
on clearStage me
— 1. Move Movie
me.moveMovie(#OffStage)
pauseTime = sprite(pSprite).movieTime
numViewings = me.addSighting(pauseTime)
— 2. Set variables
pAvailable = #Available
pMovieTitle = 0
sprite(pSprite).movieRate = 0
```

```
sprite(pSprite).member = VOID
return [pauseTime, numViewings]
end
— ============================================
on addSighting me, pauseTime
myDuration = sprite(pSprite).member.duration — in ticks
thisViewing = the timer - pStartTime — in ticks
if (thisViewing + pOldPauseTime) > myDuration then numViewings = 1
else numViewings = 0
return numViewings
end
— ============================================
on moveMovie me, onOrOffStage
case (onOrOffStage) of
#OnStage:
sprite(pSprite).rect = pStageArea
updateStage
#OffStage:
sprite(pSprite).rect = pOtherArea
updateStage
end case
end
— ============================================
— The following handlers read/write the variables.
— ============================================
on getStatus me
return pAvailable
end
```

When the Video Channel object is instantiated, it creates two rectangles: pStageArea, and pOtherArea. When a movie begins playing, its sprite is then moved to pStageArea; when it's to be stopped, its sprite is first moved to pOtherArea before it's stopped. This prevents a flash when the movie starts or stops. findViewableArea() and findOtherArea() define these two rectangles.

setStage() and clearStage() handle the playing of the QuickTime movie. As the comments state, setStage() starts playing the movie and then moves the sprite on-stage after the playback head enters the next frame of the Score. It sets a series of variables and sends a message to playAlternateTrack(), described below, to make sure that the original sound track is playing.

clearStage() essentially performs the reverse operation: it moves the movie off-stage, stops playing it, sets the appropriate variables, and then returns information about the play state of the movie to the calling function (the Video Mixer), which writes this information to the media object. As stated previously, the Video Channel object does not deal directly with the object, and so it does not have the capability of writing the pause time or view list information into the media object.

The following group of handlers deal with input from the Bed sensors.

```
— ********************************************************************************
— Everything from this point on determines how the Video Channel object responds to input from the Bed sensors.
— ********************************************************************************
— ============================================
— recdFrameEvent() is called by the frame script that loops the QT movie sprites.
— ============================================
on recdFrameEvent me
case (pLoopState) of
0: nothing
1:
if sprite(pSprite).movieTime > pLoopTime then
sprite(pSprite).movieTime = pLoopTime - pLoopLength
else
nothing
end if
```

```
end case
end
— ============================================
on loopMovie me, percentChange
percentTrigger = 60
if percentChange > percentTrigger then
if pLoopState = 0 then
pLoopState = 1
pLoopTime = sprite(pSprite).movieTime
else
nothing
end if
else
pLoopState = 0
end if
end
— ============================================
on changeMovieRate me, whichDirection, percentChange
minRate = 0.1
normRate = 1
maxRate = 2
case (whichDirection) of
#Fast:
newRate = 1 + ( ( float( percentChange ) * ( maxRate - normRate ) ) / 100 )
if newRate < 1.2 then newRate = 1
#Slow:
newRate = 1 - ( ( float( percentChange ) * ( normRate - minRate ) ) / 100 )
if newRate > .80 then newRate = 1
end case
sprite(pSprite).movieRate = newRate
end
— ============================================
— playAlternateTrack() switches from sound track in the QT movie to another.
— playAlternateTrack() works as follows:
— a. It divides 100 by the number of currently-playing channels, to get a percentage value per channel;
— b. It maps the percentValue (passed as an argument) to a channel;
— c. It plays the channel;
— d. It mutes the other channels.
— ============================================
on playAlternateTrack me, percentValue
audTracks = 0
repeat with n = 1 to sprite(pSprite).member.trackCount()
if sprite(pSprite).trackType(n) = #sound then
audTracks = audTracks + 1
else
nothing
end if
end repeat
if audTracks > 1 then
percentPerChannel = 100/audTracks
pQTAudioTrack = (percentValue/percentPerChannel) + 1 — This is to round up, not down
if pQTAudioTrack > audTracks then pQTAudioTrack = audTracks
me.soloTrack(pQTAudioTrack, audTracks)
else
nothing
end if
end
```

```
on soloTrack me, channelNum, audTracks
repeat with n = 1 to audTracks
thisAudTrack = n + 1 — The first track is video.
if n = channelNum then
sprite(pSprite).setTrackEnabled(thisAudTrack, TRUE)
else
sprite(pSprite).setTrackEnabled(thisAudTrack, FALSE)
end if
end repeat
end
— ===========================================
— changeVolume() is called by the Bed scripts.
— ===========================================
on changeVolume me, percentRate
— maxVolume = 255
maxVolume = 200 — 6/04: we noticed crackling in the audio, and so we're backing down the max volume.
sprite(pSprite).volume = maxVolume - ( ( float(percentRate)/100.0 ) * maxVolume )
end
```

As noted earlier, recdFrameEvent() is the real-time event posting that allows the Video Channel object to keep track of the progress of the movie. As of June 2001, the only function that takes advantage of this is looping: by keeping track of the current play time of the movie, we can implement a loop when a participant pushes on a Bed sensor. Setting the loop occurs in two stages: the loopMovie() handler sets a flag, and then recdFrameEvent() implements the loop.

changeMovieRate() responds to two different sensors on the Bed, one of which slows the movie down and the other of which speeds the movie up.

playAlternateTrack() and soloTrack() work together to change the QuickTime movie's sound track. playAlternateTrack() looks through all the tracks of the QuickTime movie and finds the audio tracks. It then determines which track to play on the basis of pressure from the Bed sensor, and then sends the information to soloTrack(), which plays that track. NOTE: soloTrack() assumes that the audio tracks are grouped and start with the second track of the movie; if a QuickTime movie has two video tracks, or if the audio tracks are not contiguous, this handler will fail.

changeVolume() does exactly that. NOTE: in our test in June 2001, we noticed a lot of digital clicking in the sound tracks, and so we're experimenting with lowering the volume to see if that helps.

NOTE: the actual code contains many more handlers, which move, mask, and resize the QuickTime movies. None of them were satisfactory, and so the Bed sensors no longer call them, but the code remains. It's not included here because it's irrelevant to the final project.

The Audio Mixer and Audio Channel Objects

Another of the lower-level display objects is the Audio Mixer. Just as with the Video Mixer, the Audio Mixer is created by two object constructors, the Audio Mixer Object Constructor and the Audio Channel Object Constructor. The architecture of the Audio Mixer is virtually identical to that of the Video Mixer, so the code is presented here without comment. The only substantial difference between them is that the Audio Mixer is dealing with sound channels in the Score, rather than sprite channels.

The Audio Mixer Object Constructor

The Audio Mixer mixes separate sound files, so that they fade smoothly during the transition from one to another. The Audio Mixer is employed to play the sound files that accompany a particle animation, and the sound files are passed in as a list of linked media titles.

The code for the Audio Mixer object is presented below, without additional comment.

```
— ===========================================
— 4/21/01. Author: David Johnson
— This script establishes the Audio Mixer object.
— gDisplayObject.pAudioMixerObject has the following functions:
— a) Coordinate and control the sound tracks.
— ===========================================
— Static variables: these are set once.
property pTotalChannels, pChannelObjectList
— State variables: these change over time.
```

```
property pCurrentChannel, pNumChannels, pDefaultChannel
property ancestor
— ===========================================
— new() performs the following:
— a. Creates the channel objects, which manage the audio channels.
— ===========================================
on new me
global gDisplayObject
— 11/27/00: the mixer object controls all the sound channels, evaluating which ones are playing and which ones are available, setting loops, etc.
me.ancestor = gDisplayObject
pTotalChannels = 8 — 11/26: this is the max number of sound channels that Director supports.
pChannelObjectList = []
repeat with n = 1 to pTotalChannels
add pChannelObjectList, new(script “Audio Channel Object Constructor”, n)
end repeat
pNumchannels = 0 — This is the number of channels actually playing audio at the moment.
return me
end
— ===========================================
— setStage handles the playing of sound files.
— NOTE: if the object calling the linked sound files is an #Im object, we need to play sound at once, since the linked media is the only available sound track. If the object is a #Mov object, it has its own sound track; these are merely alternates.
— setStage() performs the following:
— a. Kills all the currently-playing sounds.
— b. Determine the calling object.
— c. Assign each member in the songList to its own channel, muting all but the foreground object (#Im) or all object (#Mov).
— d. Setting pNumChannels, so that we know how many alternate tracks there are.
— ===========================================
on setStage me, songList, mediaType
repeat with n = 1 to pChannelObjectList.count
pChannelObjectList[n].clearStage(#Medium, #Kill)
end repeat
if songList <> 0 then
if mediaType = #Im then
pCurrentChannel = findAvailableChannel() — 4/21: if clearStage() worked, this should presumably return 1.
else
pCurrentChannel = 0
end if
repeat with n = 1 to songList.count
if n = pCurrentChannel then
pChannelObjectList[n].setStage(songList[n], #Medium, #Fore)
else
pChannelObjectList[n].setStage(songList[n], #Medium, #Back)
end if
end repeat
pNumChannels = songList.count
pDefaultChannel = pCurrentChannel
if pDefaultChannel = 0 then pDefaultChannel = 1
else
nothing — The Video Mixer handles this.
end if
end
— ===========================================
— findAvailableChannel() queries the channel objects, looking for an available one.
— ===========================================
on findAvailableChannel me
```

```
repeat with n = 1 to pChannelObjectList.count
if pChannelObjectList[n].getStatus() = #Available then return n
end repeat
return 0 — This will be returned if there are no available tracks; otherwise, “return n” above will terminate the function.
end
— ============================================
— changeVolume() changes the system volume, from 1 to 7. this is called by the Bed sensors.
— ============================================
on changeVolume me, percentVolume
repeat with n = 1 to pChannelObjectList.count()
pChannelObjectList[n].setCurrentVolume(percentVolume)
end repeat
pChannelObjectList[pCurrentChannel].changeVolume()
end
— ============================================
— playAlternateTrack() switches from one linked sound track to another.
— playAlternateTrack() works as follows:
— a. It divides 100 by the number of currently-playing channels, to get a percentage value per channel;
— b. It maps the percentValue (passed as an argument) to a channel;
— c. It plays the channel;
— d. It mutes (but does not kill) the other channels.
— ============================================
on playAlternateTrack me, percentValue
if pNumChannels > 0 then
percentPerChannel = 100/pNumChannels
channelNum = (percentValue/percentPerChannel) + 1 — This is to round up, not down
if channelNum > pNumChannels then channelNum = pNumChannels
repeat with n = 1 to pNumChannels
if n = channelNum then
pChannelObjectList[n].resumePlay(#Short)
pCurrentChannel = n
else
pChannelObjectList[n].clearStage(#Short, #Mute)
end if
end repeat
else
nothing
end if
end
— ============================================
— The following handlers offer support services:
— getSoundChannel() returns the current foreground channel.
— ============================================
on getSoundChannel me, whichChannel
case (whichChannel) of
#current: return pCurrentChannel
otherwise: put “MixerObject.getSoundChannel(): I don’t recognize the sound channel.”
end case
end
```

The Audio Channel Object Constructor

As with the Audio Mixer, the Audio Channel object is so similar to the Video Channel object that the code is presented here without additional comment.

```
— ============================================
— 4/21/01. Author: David Johnson
— This script establishes the Track objects.
— gDisplayObject.pAudioMixerObject.pChannelObjectList[whichTrack] has the following functions:
```

```
— a) Handle the media and volume of each sound track.
— ============================================
property pChannel, pMember — Static variables: these are unique to each track and are set at initialization.
property pAvailable — State variables: this is unique to each track and changes continually.
property pLoopCount, pQuietVolume, pLoudVolume, pCurrentVolume, pLongFade, pMediumFade, pShortFade — Constants: these are the same for every track.
— ============================================
— new() performs the following:
— a. Assigns all the constant and static variables except pMember (which is set by setStage(), below);
— b. Announces the track is available to play a sound file by setting pAvailable to #Available;
— c. Mutes the track.
— ============================================
on new me, whichChannel
— Static variables.
pChannel = whichChannel
— Constants.
pLoopCount = 100
pQuietVolume = 1 — 11/26: I've seen this reset itself to 255 if its value is set to zero, so I set it to 1.
pLoudVolume = 255
pCurrentVolume = pLoudVolume
pLongFade = 4000 — in milliseconds
pMediumFade = 1000 — in milliseconds
pShortFade = 500 — in milliseconds
— State variables.
pAvailable = #Available — This means that the Audio Mixer can send me a sound file to play. The states are: #Available, #Mute, and #Playing.
— Mute the track.
sound(pChannel).volume = pQuietVolume
return me
end
— ============================================
— setStage(), resumePlay, and clearStage() handle the playing of sound files.
— setStage() performs the following:
— a. Begins playing the member;
— b. If called to be the foreground channel, fades the channel up and sets pAvailable to #Playing; if called to be the background, sets pAvailable to #Mute and keeps the channel muted;
— c. Sets the member name.
— resumePlay() performs the following:
— a. Fades the channel up and sets pAvailable to #Playing.
— clearStage() performs the following:
— a. Fades the channel down;
— b. If called to run in #Mute mode, sets pAvailable to #Mute and keeps the channel muted; if called to stop playing, sets pAvailable to #Available and deletes the member reference.
— c. If called to stop playing, calls killChannel().
— ============================================
on setStage me, memberName, fadeTime, backOrFore
— 1. Start playback
case (backOrFore) of
#Fore:
— Fade in
sound(pChannel).play(member(memberName))
timeOfFade = me.getFadeTime(fadeTime)
sound(pChannel).fadeTo(pCurrentVolume, timeOfFade)
— Set variables
pAvailable = #Playing
#Back:
— Set variables
```

```
sound(pChannel).stop()
sound(pChannel).queue(member(memberName))
pAvailable = #Mute
end case
pMember = memberName
end
— ============================================
on resumePlay me, fadeTime
timeOfFade = me.getFadeTime(fadeTime)
sound(pChannel).play()
sound(pChannel).fadeTo(pCurrentVolume, timeOfFade)
pAvailable = #Playing
end
— ============================================
on clearStage me, fadeTime, muteOrKill
— 1. Fade out
if sound(pChannel).volume <> pQuietVolume then
timeOfFade = me.getFadeTime(fadeTime)
sound(pChannel).fadeTo(pQuietVolume, timeOfFade)
end if
— 2. Set variables
case (muteOrKill) of
#Mute:
sound(pChannel).pause()
pAvailable = #Mute
#Kill:
pAvailable = #Available
pMember = 0
sound(pChannel).stop()
end case
end
— ============================================
— killChannel() frees up sound channels that are no longer being used.
— ============================================
on killChannel me
if ( pAvailable = #Available ) and ( sound(pChannel).volume = 1 ) then
if sound(pChannel).isBusy() then sound(pChannel).stop()
end if
end
— ============================================
— The following two handlers are run in response to a volume-change request from the Bed sensors.
— setCurrentVolume sets a value for the current volume, based on the pressure on the sensor. It doesn't change the audible volume.
— changeVolume() changes the volume.
— ============================================
on setCurrentVolume me, percentVolume
pCurrentVolume = integer( pLoudVolume - ( pLoudVolume * ( float (percentVolume) / 100.0 ) ) )
end
on changeVolume me
sound(pChannel).volume = pCurrentVolume
end
— ============================================
— The following handlers offer support services:
— getFadeTime() converts a general directive, passed as a parameter, to milliseconds.
— ============================================
on getFadeTime me, fadeTime
case (fadeTime) of
#Short: timeOfFade = pShortFade
```

```
#Medium: timeOfFade = pMediumFade
#Long: timeOfFade = pLongFade
end case
return timeOfFade
end
— ============================================
— The following handlers read/write the variables.
— ============================================
on getStatus me
return pAvailable
end
```

The Particle Object

The last of the lower-level objects directly responsible for displaying media is the Particle object. Although the Particle object is also created through two distinct objects, the architecture is quite different from that of the Video and Audio mixers. The Particle object is itself a property of the Display object, and it stores relevant information about the current particle animation such as member names and Bed sensor properties—to that extent, it's similar to the Video and Audio mixers. However, the second object is not a property of the Display object. Instead, it's an independent behavior, which is attached to the sprites on the stage. This behavior calls the Particle object to load the appropriate variables, which the behavior then uses to affect the appearance of the sprites.

Using Lingo to create particle systems was an idea that we first encountered in an article by Charles Forman on the Director Online web site. In his examples, he used solid colors as the basis for his particle animations; we adapted the idea to generate animations from images. We have adapted some of Charles Forman's code here; the rest is our own creation. The original article is available at www.director-online.com; more of Mr. Forman's work is available at his web site, www.cforman.com.

The Particle Object Constructor

The Particle object is created when the Display object is initialized.

```
— ============================================
— 4/17/01. Author: David Johnson
— Adapted from code written by Charles Forman (www.cforman.com) for www.director-online.com
— This script establishes the Particle object.
— gDisplayObject.pParticleObject has the following functions:
— a. Create a particle system display from the images.
— b. Handle effects that are applied to image groups (e.g., movement, rate, tint, etc.)
— c. Hold property variables that the particle sprites can call for their values.
— ============================================
property pMaxOpacity, pCurrentParticleColor, pOriginalParticleColor, pBaseMemberList, pNewMemberList, pOriginPoint, pSpread,
pAngle, pBlendMomentum, pMomentum, pMomentumDampen, pFireType, pAlphaMemName, pLogMemberName
global gDisplayObject
— ============================================
— new() initializes some of the property variables. Others are initialized when setStage() is called by gDisplayObject.
— ============================================
on new me
pMaxOpacity = 100
pCurrentParticleColor = rgb(0, 0, 0)
pOriginalParticleColor = 0
pBaseMemberList = []
pNewMemberList = []
pOriginPoint = [320, 240]
pLogMemberName = "Particle Log"
return me
end
— ============================================
— setStage() loads up the media object and begins the process of displaying it. The sprites themselves take over, calling getSpriteVals(),
below, to load their properties from the Particle object's variables.
— setStage() performs the following:
— 1. Get the members that serve as the basis for the media that will be displayed. "Basis" is IMPORTANT: we don't want to distort or
```

```
manipulate the original media, so we duplicate it (hence, pNewMemberList vs. pBaseMemberList).
— 2. Initialize pNewMemberList if it isn't already. This ensures that we can erase cast members when we call createFireImages().
— 3. Call loadUpPresets() to create properties for the sprites.
— 4. Call createFireImages() to create cast members for the sprites.
— 5. Call addSighting() so that the object increments its view list.
— ==========================================
on setStage me, newMediaObject
pBaseMemberList = newMediaObject.getMemberList()
— Set an initial length for pNewMemberList
if pNewMemberList.count = 0 then
repeat with n = 1 to pBaseMemberList.count
add pNewMemberList, 0
end repeat
end if
— Load up preset (e.g., #Oil)
pFireType = me.getRandomPreset()
me.loadUpPresets(pFireType)
pOriginalParticleColor = newMediaObject.getColor()
pCurrentParticleColor = duplicate(pOriginalParticleColor)
— Create cast members
me.createFireImages()
newMediaObject.addSighting()
end
— ==========================================
— createFireImages() creates the cast members for the sprites.
— createFireImages() performs the following:
— 1. Create tempList, a buffer to hold the cast members.
— 2. Create a series of #bitmap cast members, marrying the images from pBaseMemberList with the alpha channel of "Particle Alpha" to
create a series of new images.
— 3. Load those cast members into tempList.
— 4. Once tempList is full, load its members into pNewMemberList.
— ==========================================
on createFireImages me
tempList = []
repeat with i = 1 to pBaseMemberList.count()
baseMember = member(pBaseMemberList[i])
— Set the alpha
BaseImageAlpha = member(pAlphaMemName, "Display Object Scripts").image
— Create new cast member, name it, make image
TargetMember = new(#bitmap, castLib "On-screen Images")
TargetMember.name = "Fire" & i
— TargetMember.image = image(member("ParticleAlpha").image.width, member("ParticleAlpha").image.height, 32)
TargetMember.image = image(member(baseMember).image.width, member(baseMember).image.height, 32)
— Duplicate the baseMember's image, to avoid overwriting the base member
originalImage = baseMember.image.duplicate()
— Fill the image with a copy of the base member
TargetMember.image.copyPixels(originalImage, TargetMember.image.rect, baseMember.image.rect, [#color: pCurrentParticleColor])
— Set alpha to nothing and make a copy of it (seems asinine, but otherwise, the resulting images alpha has noise (BUG??))
TargetMember.image.setAlpha(0)
emptyAlpha = targetMember.image.extractAlpha()
newAlpha = emptyAlpha
newAlpha = duplicate(emptyAlpha)
— Use copyPixels to apply the ParticleAlpha to the copied alpha
newAlpha.copyPixels(BaseImageAlpha,BaseImageAlpha.rect,BaseImageAlpha.rect)
— Set the Alpha
targetMember.image.setAlpha(newAlpha)
targetMember.useAlpha = true
```

```
add tempList, targetMember
end repeat
pNewMemberList = tempList
end
— ============================================
— These handlers create various looks for the cast members.
— loadUpPresets() creates values for the sprite properties.
— getRandomPreset() grabs one of the styles at random, to give us some variety.
— ============================================
on loadUpPresets me, preset
case (preset) of
#Oil: presetArray = [55,195,18,6,1.00]
#Flame: presetArray = [12,0,34,10,.90]
#FieldFire: presetArray = [360,0,33,10,1.00]
#RandomFire:
presetArray = [random(360), random(360), random(17)+17, random(10)+5, random(4.00)/10.00 + .6]
end case
AlphaMemName = “ParticleAlpha” & random(4)
pSpread = presetArray[1]
pAngle = presetArray[2]
pBlendMomentum = presetArray[3]
pMomentum = presetArray[4]
pMomentumDampen = presetArray[5]
end
on getRandomPreset end
presetList = [#Oil, #Flame, #FieldFire, #RandomFire]
return presetList[ random(4) ]
end
— ============================================
— The sprites call getSpriteVals() to set their own properties.
— ============================================
on getSpriteVals me
randomMember = random(pNewMemberList.count())
valList = [pMaxOpacity, pOriginPoint, pMomentum, pSpread, pAngle, pBlendMomentum, pMomentumDampen,
pNewMemberList[randomMember]]
return valList
end
— ============================================
— These handlers are manipulated by the Bed sensors.
— setOriginPoint() and getOriginPoint() allow the Bed sensors to move the Particle system.
— changeParticleColor() changes the RGB value of the images.
— setOpacity() changes the opacity of the images.
— ============================================
on setOriginPoint me, percentagePoint
maxX = 640
minX = 320
maxY = 480
minY = 240
Origin = (percentagePoint * 640) / 100
yOrigin = (percentagePoint * 480) / 100
if xOrigin > maxX then xOrigin = maxX
if xOrigin < minX then xOrigin = minX
if yOrigin > maxY then yOrigin = maxY
if yOrigin < minY then yOrigin = minY
pOriginPoint = [xOrigin,yOrigin]
end
on getOriginPoint me
```

```
return pOriginPoint
end
— ============================================
on changeParticleColor me, whichAmount
colorDelta = 20
changeOrNot = 1
newRed = pOriginalParticleColor.red * (float(whichAmount)/100)
newGreen = pOriginalParticleColor.green * (float(whichAmount)/100)
newBlue = pOriginalParticleColor.blue * (float(whichAmount)/100)
if newRed > colorDelta then changeOrNot = 1
else if newGreen > colorDelta then changeOrNot = 1
else if newBlue > colorDelta then changeOrNot = 1
case (changeOrNot) of
0: nothing
1:
pCurrentParticleColor.red = newRed
pCurrentParticleColor.green = newGreen
pCurrentParticleColor.blue = newBlue
me.deleteFireImages()
me.createFireImages()
end case
end
— ============================================
on setOpacity me, whichAmount
maxOpacity = 100
minOpacity = 30
pMaxOpacity = maxOpacity - ( ( maxOpacity - minOpacity ) * ( float (whichAmount) / 100.0) ) — inverted
end
— ============================================
— Testing. We print out particularly good examples to use as models.
— ============================================
on logGoodExample me
put “The number of seconds is” && (the timer)/60 & RETURN after member(pLogMemberName)
put “pSpread is” && pSpread & RETURN after member(pLogMemberName)
put “pAngle is” && pAngle & RETURN after member(pLogMemberName)
put “pBlendMomentum is” && pBlendMomentum & RETURN after member(pLogMemberName)
put “pMomentum is” && pMomentum & RETURN after member(pLogMemberName)
put “pMomentumDampen is” && pMomentumDampen & RETURN after member(pLogMemberName)
put “Alpha Member:” && pAlphaMemName & RETURN after member(pLogMemberName)
put “Image members:” && pBaseMemberList & RETURN after member(pLogMemberName)
put RETURN & RETURN after member(pLogMemberName)
end
— ============================================
— Housekeeping. This prevents unused cast members from proliferating.
— ============================================
on cleanUp me
me.deleteFireImages()
end
on deleteFireImages me
repeat with i = 1 to pNewMemberList.count
member(pNewMemberList[i]).erase()
end repeat
repeat with n = 1 to the number of members of cast “On-Screen Images”
if member(n, “On-Screen Images”).name contains “Fire” then
alert(“Bad member:” && member(n, “On-Screen Images”).name)
else
nothing
```

Title: Dream**Life** 06.01 Medium: Installation Dimensions: 27ft x 39ft Date: June 2001

```
end if
end repeat
end
```

The Sprite Behavior

The behavior is described below.

```
— ============================================
— 4/17/01. Author: David Johnson
— Adapted from code written by Charles Forman (www.cforman.com) for www.director-online.com
— This behavior establishes one sprite for the particle system.
— ============================================
property pMaxOpacity, pSpread, pAngle, pBlendMomentum, pMomentum, pMomentumDampen, pPointX, pPointY, pSpinSpeed,
pRadius, pScale, pMemberName, pSpriteBlend
global gDisplayObject
on beginSprite me
me.getValuesFromPartObject()
— Set up a random spin of the particle
pSpinSpeed = Random(60)-30
— Set the initial distance from the originating point
pRadius = 5
— Set the initial blend of the particle to 0
pSpriteBlend = 0
pScale = 0
— Set a member for the particle
sprite(me.spriteNum).member = Member(pMemberName)
end
on getValuesFromPartObject me
valList = gDisplayObject.getSpriteVals()
— Set the peak opacity of the particle
pMaxOpacity = valList[1]
— Set the originating point of the particle
pPointX = valList[2][1]
pPointY = valList[2][2]
— Set up a loosely random inital momentum
pMomentum = Random(valList[3])+25
— Set up a loosely random direction the particle should go in
pAngle = Random(valList[4])+ valList[5]
— Set the momentum of the blend
pBlendMomentum = valList[6]
pMomentumDampen = valList[7]
pMemberName = valList[8]
end
on exitFrame me
— Dampen the momentum
pMomentum = pMomentum * pMomentumDampen
— Apply the momentum to the radius
pRadius = pRadius + pMomentum
— Apply the momentum to the scale
pScale = pScale + pMomentum
— Apply the blend momentum to the blend of the sprite
pSpriteBlend = pSpriteBlend + pBlendMomentum
— Set all sprite properties
sprite(me.spritenum).width = pScale
sprite(me.spritenum).height = pScale
sprite(me.spritenum).rotation = sprite(me.spritenum).rotation + pSpinSpeed
— Make sure the sprite's blend is always within 0 and 100
sprite(me.spritenum).blend = min(max(pSpriteBlend, 0), 100)
— Calculate the position of the sprite based on the angle and radius from the originating point
```

```
sprite(me.spritenum).locH = pRadius*cos(pAngle*pi()/180) + pPointX
sprite(me.spritenum).locV = pRadius*sin(pAngle*pi()/180) + pPointY
— As the particle begins it fades in from 0. However when the blend reaches pMaxOpacity, then
— the particle begins to fadeout at a random rate
if pSpriteBlend > pMaxOpacity then pBlendMomentum = -(Random(16) + 4)
— When the partcle fades out completely, recycle the sprite
if pSpriteBlend < 0 then BeginSprite me
end
```

The Bed Sensor Object

The Video Mixer object, the Audio Mixer object, and the Particle object are the three sub-objects of the Display object that are directly responsible for displaying media. There is one more sub-object of the Display object, which manages the Canopy movie's responses to messages coming in from the Bed, much as the Logic object manages the Canopy movie's responses to messages coming in from the Book and Radio.

As with the other sub-objects, the Bed Sensor object is created in two parts: the Bed Sensor object is initialized, and then it instantiate a series of objects and populates a list with those objects, each of which is responsible for a single sensor on the bed. Each of those objects has a handler that determines how the Canopy movie responds when a QuickTime video is playing, and another that manages the response when a particle animation is playing.

The Bed Sensor Object Constructor

The Bed Sensor object is created when the Display object is initialized.

```
— ============================================

— 5/24/01. Author: David Johnson

— This script establishes the Bed Sensor object.
— gDisplayObject.pBedSensorObject has the following functions:
— a. Store the objects that interprest the bed sensor input in a list.
— b. Perform the calculations that allow us to interpret average input from the bed.
— c. Alter the messages from the Bed to the Display object if they fall within the moving average.
— ============================================
property pTempList, pAverageList — These are the lists that store the immediate and averaged values.
property pDelta, pAvTime, pSampleInterval, pNormList, pNormRate, pLastTime — These are the variables used to calculate the moving
averages.
property pBedSensorList
on new me
pTempList = []
pAverageList = []
pNormList = [] — The number that's subtracted from the Temp value to return a normalized value
numOfSensors = 6
repeat with n = 1 to numOfSensors
add pTempList, 0
add pAverageList, [[0, 0], [0, 0], [0, 0]]
add pNormList, 0
end repeat
pDelta = 10 — The difference between the Average and Temp values.
pAvTime = 5 * 60 — in ticks
pSampleInterval = 1 * 60 — in ticks
pNormRate = 5 — The number that's added to pNormList; higher numbers = faster normalization
pLastTime = the timer
pBedSensorList = []
repeat with sensorNum = 1 to numOfSensors
newSensorName = "Sensor Object Constructor" && sensorNum
add pBedSensorList, new(script newSensorName)
end repeat
return me
end
```

```
— ##########################################3
—
— BED SENSOR HANDLERS
—
— ============================================
— The following handlers determine how the Display object responds to input from the Bed sensors.
— runAction() is the handler that responds to input from the bed. This is in direct response to the Mailbox object.
— The arguments for runAction() are:
— a. newSubject (which should always be #bed);
— b. newContent, in the form of a linear list. The first value is the stream that the input came from (on a scale of 0 to 5), and the second value is the "voltage" reading from that stream (on a scale from 1 to 100). For example, a list of [1, 20] means that stream 1 is sending a voltage of 20%.
— switchSensorMedia() tells the sensor objects whether a #Mov or #Im object is being displayed. The sensors may act differently as a result. The handler is called by setStage(), above.
— ============================================
on runAction me, newSubject, newContent
case (newSubject) of
#bed:
sensorObject = pBedSensorList[newContent[1] + 1] — The messages are zero-indexed; Lingo lists are one-indexed.
sensorObject.runAction(newContent[2])
otherwise: nothing — 4/8: this should only be from the bed.
end case
end
on switchSensorMedia me, mediaType
repeat with n = 1 to pBedSensorList.count()
pBedSensorList[n].changeMediaType(mediaType)
end repeat
end
— ##########################################3
—
— MOVING AVERAGE HANDLERS
—
— ============================================
— The following handlers provide input to the Bed Sensor object.
— recdFrameEvent triggers this routine at fixed times. This allows the Bed Sensor object to fill in intervals during which we're not receiving input from the Bed.
— getInterpretedAction receives input from a sensor, interprets it, and returns the interpreted value.
— ============================================
on recdFrameEvent me
if (the timer) > (pLastTime + pSampleInterval) then
repeat with n = 1 to pAverageList.count()
sensorObject = pBedSensorList[n]
sensorObject.runAction(pTempList[n])
end repeat
pLastTime = the timer
else
nothing
end if
end
on getInterpretedAction me, whichIndex, whichValue
pTempList[whichIndex] = whichValue
newValue = me.evalSample(whichIndex, whichValue)
return newValue
end
— ============================================
— The following handlers determine how the Display object averages input.
— For each sensor, evalSample() performs the following:
```

— a. It adds a linear list with the time stamp and value (e.g., [3600, 40], which indicates a value of 40 was read after one minute);
— b. It calls deleteOldValues() to cull anything older than pAvTime from the list;
— ==========

```
on evalSample me, whichIndex, whichValue
xyList = []
add xyList, the timer
add xyList, whichValue
me.deleteOldValues(whichIndex)
add pAverageList[whichIndex], xyList
averageValue = me.calculateAverage(whichIndex)
if abs(whichValue - averageValue) > pDelta then
pNormList[whichIndex] = 0
else
if (pNormList[whichIndex] + pNormRate) < whichValue then
pNormList[whichIndex] = (pNormList[whichIndex] + pNormRate)
else
pNormList[whichIndex] = whichValue
end if
end if
newValue = whichValue - pNormList[whichIndex]
return newValue
end
on deleteOldValues me, whichIndex
repeat with n = 1 to pAverageList[whichIndex].count()
newRecord = pAverageList[whichIndex][n]
if newRecord[1] < ( the timer - pAvTime) then
pAverageList[whichIndex].deleteAt(n)
else
nothing
end if
end repeat
end
on calculateAverage me, whichIndex
sum = 0
repeat with n = 1 to pAverageList[whichIndex].count()
newRecord = pAverageList[whichIndex][n]
sum = sum + newRecord[2]
end repeat
return sum/ ( pAverageList[whichIndex].count() )
end
```

Sensor Object
The bed sensors perform the following functions:
Sensor
When Particle system is playing
When QuickTime is playing
0 (upper right)
Tints the images
Slows down the movie
1 (upper middle)
Changes volume
Loops the movie
2 (upper left)
Selects an alternate sound track
Speeds up the movie
3 (lower right)
Changes the opacity of the images

Selects an alternate sound track
4 (lower middle)
Changes the frame rate (slows down the particle animations)
Loops the movie (same as #1)
5 (lower left)
Moves the origin point of the animations
Changes volume
The code for the first sensor, #0, is shown below.

```
— ============================================
— 4/17/01. Author: David Johnson
— This is a sensor object. It stores the handlers that define how the Canopy movie responds to one Bed sensor.
— ============================================
property pMediaType, pAverage, pLastValue, pValueDelta
global gDisplayObject
on new me
pAverage = 1 — 5/24: default value: this sensor CAN be averaged out
pValueDelta = 10
pLastValue = -1
return me
end
— ============================================
— changeMediaType() tells this sensor object that a different media type is being played.
— ============================================
on changeMediaType me, whichType
pMediaType = whichType
pLastValue = -1
case (whichType) of
#Mov: pAverage = 1
#Im: pAverage = 1
end case
end
— ############################################3
—
— MOVIE/PARTICLE HANDLERS
—
— ============================================
— runAction() calls the actions associated with this sensor.
— ============================================
on runAction me, whichAction
mySensor = 1
if pAverage = 1 then
interpretedAction = gDisplayObject.pBedSensorObject.getInterpretedAction(mySensor, whichAction)
else
interpretedAction = whichAction
end if
if (pLastValue = -1) OR ( abs( pLastValue - interpretedAction ) > pValueDelta ) then
pLastValue = interpretedAction
case (pMediaType) of
#Mov: me.changeMovieRate(#Slow, interpretedAction)
#Im: me.tintImages(interpretedAction)
end case
```

```
else
nothing
end if
end
— ==========================================
— changeMovieRate() changes the rate that the QT movie displays.
— ==========================================
on changeMovieRate me, whichDirection, whichAction
gDisplayObject.changeMovieRate(whichDirection, whichAction)
end
— ==========================================
— tintImages() tints the image group.
— ==========================================
on tintImages me, whichAction
gDisplayObject.changeParticleColor(whichAction)
end
```

Budget

Budget Overview

DreamLife was not expensive. According to the NAVE group from Georgia Tech who presented last year at SIGGRAPH2000, an "inexpensive" installation goes for under $30,000.00. Our group prides itself on the fact that, through class fees, we had a total budget of $1,100.00 at the beginning of the 2000-2001 academic year and have managed to stay, if not within that amount, very close to it.

Project Assessment

Evaluation Criteria

Our criteria for success.
Is the narrative engaging?
Is the experience immersive?
Does it explore new way of interacting with digital media and each other?
Does it push the boundaries of technology? In a way that is emotionally and intellectually stimulating?
Is the message meaningful? Is the message clear?
Does the art support the concept?
Does the physical space complement the digital media?
Aesthetically, are these the best combinations of imagery/sound/video/text?
Is the integration of media successful?
Is it interesting the first time, the second time, the 10th time?
Does the project exhibit technical proficiency?
Is it ambitious enough?

Our Goals

Our personal goals for the project are simple. We've discussed the projects we've seen (in our course work, in the forum, and outside of school) and have found that many of them may be technologically advanced or economically viable, but only a very few have been artistic. Of these, some (such as the Memarena project, which is listed in our bibliography) are intellectually appealing, but it's hard to imagine (at least from the description on its web page) that its participants found it emotionally compelling. If we are truly participating in the birth of a new medium, we want to discover the work that is compelling within that medium. With this in mind, our goals can be summarized as follows:

Our project should be inspiring; that is, it should be both intellectually and emotionally stimulating
We want to push boundaries.
We want to explore multi-user interaction.
We want to provide a community space. Even the best projects we've seen (such as Osmose) are essentially single-user experiences.
We want to explore multi-user interaction.
In order to reach our goal of pushing boundaries we will need to address some fundamental questions.
What is digital media?
What are the current methods/conventions for interacting with digital media?

How can we challenge these conventions?
Our overall goal is to create a space that is engaging. We will assess this through a series of run-throughs and critiques. Is our message clear? Is the artwork inspiring? Are we pushing the limits of technology to create something beautiful?

Conclusion
Without being overly self-congratulatory, we must say that we not only met each and every one of our goals, but we exceeded all of our expectations. After the premiere on June 6, we met and evaluated the feedback we received from participants in the installation. We ran over a copy of the goals stated above, and found that our answer in every case was "yes": we pushed boundaries, we explored multi-user interaction, our experience was immersive, our narrative was engaging... This is not only our opinion; it was borne out repeatedly from the comments we heard from the participants in the room. Perhaps the best summation was this one comment, overheard in the room:

"How did you get access to this woman's dreams?"

Bibliography

This is a list of some of the books, articles, films, and web sites we have found, as well as some of the installations and panels we've attended.

Books and Short Stories:

Alvarez, A., Night, W. W. Norton and Company, New York, NY 1995
Birren, Faber, Color and Human Response, Van Nostrand Reinhold Company, New York, NY 1978
Capra, Fritjof, The Tao of Physics, Shambhala Publications, Inc., Boston, MA 1983
Cunningham, Scott, Sacred Sleep: Dreams and the Divine, The Crossing Press, Freedom, CA 1992
Delaney, Gayle, All About Dreams, Harper San Francisco, San Francisco, CA 1998
Deutsch, David, The Fabric of Reality, Penguin Books, New York, NY, 1997
Evan, Christopher, Landscapes of the Night, The Viking Press, New York, NY 1983
Jung, Carl Gustave. Man and His Symbols. Laureleaf, New York, NY 1964
LaBerge, Stephen and Rheingold, Howard, Exploring the World of Lucid Dreaming, Ballentine Books, New York, NY 1990
Levy, Pierre. Collective Intelligence. Perseus Press, Cambridge, MA 1997.
Murray, Janet Horowitz. Hamlet on the Holodeck. MIT Press , Cambridge, MA, 1997.
Sagan, Carl, The Dragons of Eden, Ballentine Books, New York, NY 1977
Schwartz, Delmore. 1937. "In Dreams Begin Responsibilities." In Contemporary American Short Stories, selected and edited by Douglas and Sylvia Angus. Faucet Crest, New York, NY 1992.
Stevens, Anthony, Private Myths, Harvard University Press, Cambridge, MA 1995

Journals:

Davis, Erik. "Terence McKenna's Last Trip." Wired 8.05 (March 2000): 197-209
Joy, Bill. "Why the Future Doesn't Need Us", Wired 8.04 (April 2000): 238-262

Films:

Capra, Frank, dir. It's a Wonderful Life. 1946
Cocteau, Jean, dir. Orpheus. 1949
Kurosawa, Akira. Rashomon. 1950
Proyas, Alex, dir. Dark City. 1998
Ramis, Harold, dir. Groundhog Day. 1993
Ward, Vincent, dir. What Dreams May Come. 1998

SIGGRAPH:

Emerging Technologies
This section lists the most engaging and inspiring installations we saw.
Musical Thinkers: New Pieces to Play. Presented by Joseph Paradiso. MIT Media Laboratory.
[http://www.media.mit.edu/resenv/tags.html]. An example of Tangible User Interfaces. Input devices are embedded into commonplace smart objects, they are small, wireless and inexpensive.
Plasm: In the Breeze. Presented by Peter Broadwell. Plasmatics Arts.
M3: T-Garden. Presented by Sponge. A beautiful and fun interactive installation. Where common actions enhance an everyday environment.
Biotica. Presented by Richard Brown. Royal College of art. An example of body interaction in a Virtual Environment.
Hame. Presented by Laura Beloff, Markus Decker. ARS Electronica Center/Futurelab. An example of an interactive installation, using wearables computing.
Text Rain. Presented by Camille Utterback, Romy Achituv. Interactive Telecommunications Program, NYU. An exploration of social interaction in an interactive installation.
Wooden Mirror 1999. Presented By Daniel Rozin. Interactive Telecommunications Program, NYU. A beautiful example of interactivity.

Panels

While at Siggraph, we attended the following panel:
Interactive Storytelling: New Genres and Directions. Moderator: Celia Pearce. Panelists: Steve Di Paola, Alex Mayhew, Janet H. Murray, Celia Pearce, Sarah Roberts, Michael Thomsen. This panel discussed issues related to interactive storytelling in Cyberspace. Interesting aspects about non-linear narrative were discussed.

Web Sites:

Video and Interactive Installations:
Bill Viola. [http://www.artmuseum.net/viola2/norealhtml/content/viola_gallery/BV07.html] Slowly Turning Narrative-Video/sound instal-

lation. This example shows a video installation, creating an engaging environment.
Vasulkas. [http://www.beatthief.com/three/woody/index.html] One of the focus points of their work is the creation of spaces and environments to present and experience video.
Memarena. [http://www.otherspace.de/pages/projects/memarena] An example of a multi-user, interactive and virtual environment.
Luc Courchesne. [http://www.din.umontreal.ca/courchesne/] Different examples of interactive video installations.

Academic projects and research:

Bush, Vannevar. 1945. "As We May Think." Available from http://www.isg.sfu.ca/~duchier/misc/vbush/vbush-all.shtml. Accessed June 28, 2000.
Hiroshi Ishii [http://tangible.www.media.mit.edu/groups/tangible/projects.html] He presents a new vision for HCI called Tangible Bits. It inspired our search for new ways of interact with digital media.
Janet Murray. [http://silver.skiles.gatech.edu/~murray/] Her work is focused on interactive narrative on digital media.

Media for future investigation

Books and short stories:

Borges, Jorge Luis. 1941. "The Garden of Forking Paths." In Labyrinths; Selected Stories and Other Writings, edited by James E. Irby and Donald A. Yates. W.W. Norton & Company, 1988.
Calvino, Italo. If on a Winter's Night a Traveler. Harcourt Brace, 1982.
Lightman, Alan. Einstein's Dreams. Warner Books, 1994.
Pavic, Milorad. Dictionary of the Khazars: A Lexicon Novel in 100,000 Words (Male and Female Editions). Vintage Books, 1989.

Film:

Low, Stephen, dir. Across the Sea of Time. 1996.

Music & Sound Bibliography

Collier, James Lincoln, The Making of Jazz - A Comprehensive History, Dell Publishing 1978
Trubitt, Rudy, Mackie Compact Mixers, Hal Leonard Publishing 1995

Non-technical, sound sources and Dream-related sites

http://www.findsounds.com - a great sound resources for *.wav and aiff files.
http://www.greatdreams.com/sounddrm.htm - A site focusing on documented aural dream phenomeno
http://www.greatdreams.com/sprtsnds.htm - Another from the same site
http://dreamstone.com

Technical

http://www.macmidimusic.com/ - A great resource relating to sound and midi on the Macintosh.
http://www.audiomulch.com - makers of AudioMulch graphical programming environment for the PC.
http://www.math.auth.gr/~axonis/studies/audio.htm - A site devoted to comparing audio compression algorithms. Great links too.

Chapter 3 - Appendix B

Although the final installation did not utilize midi, it was considered and pursued as a viable option for the sound station for some time. In the course of that work, I developed a number of bookmarks for midi and non-midi sites.

http://www.audiomulch.com/links.htm
http://www.sonicspot.com/
http://www.sonicspot.com/guide/glossary.html#sampler http://www.midiweb.com/hww/midi.htm
http://www.soundrangers.com/store/index.cfm
http://www.opcode.com/
http://www.audiosynth.com
http://www.audiosynth.com/schtmldocs/index.html
http://www.cassiel.com/gearhead/max.html
http://www.indiana.edu/~emusic/MIDI.html#Interactive
http://www.indiana.edu/~emusic/MIDI.html
http://www.emusician.com/
http://gigue.peabody.jhu.edu/~ich/sc/
http://www.audiomulch.com/
http://www.smarthome.com/8249.html
http://www.binaural.com/binfaq.html
http://www.sfu.ca/sca/Manuals/147/SoundEdit16/SoundEdit16.html http://www.binaural.com/

[1] Davies, Char - Technical Forum, California State University, Hayward, Spring Quarter 2000

[2] Pair, Jarrell - a quote taken from his lecture at SIGGRAPH 2000 entitled “The NAVE: Design and Implementation of a Non-Expensive Immersive Virtual Environment”, New Orleans, July 26, 2000

[3] http://www.levity.com/alchemy/atl1-5.html