XML-Driven Real-time Interactive Virtual Environment (XDRIVE) Engine

Thomas W. Corbett III

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Architecture

June 30, 2006
Blacksburg, Virginia

Keywords:
Virtual environments, interactive multimedia, XML, web-based instructional delivery

XML-Driven Real-time Interactive Virtual Environment (XDRIVE) Engine

Thomas W. Corbett III

ABSTRACT

The XDRIVE engine is a runtime solution for the coordination and display of web-based multimedia presentations that feature three-dimensional content. This 3D content is rendered in real-time, which facilitates user-defined navigation and interaction with objects contained within the 3D virtual environment. These presentations can run independently, or they can be synchronized with audio and video files.

As web browsers interpret HTML formatted files, XDRIVE presentations are authored in and interpreted from XML formatted files, which are loaded and interpreted by the engine to display the defined content. Just as web browsers can load and display external files as guided and linked by the HTML tags, XDRIVE presentations rely on links to external files that are imported and displayed as guided by the XML tags.

Developed using Macromedia Director MX – a multimedia development software package - the XDRIVE engine itself is a Shockwave file that is embedded in a web page. Shockwave, a format whose browser plug-in is free to install and is loaded on a variety of systems, allows for the coordination of multiple media and data types, and features a powerful set of tools for the use of 3D content through the Shockwave3D format.

XDRIVE is designed to open the functionality of web-based 3D to a wider audience – allowing for custom presentations to be authored without a prerequisite knowledge of complicated programming languages, and 3D scripting. The XDRIVE engine is a series of scripted systems that utilize and connect various components of Director, and provide additional capabilities above those that already exist.

ACKNOWLEDGEMENTS

It has been a very long road – without the support of my friends and family, I would never have been able to make it this far. The encouragement they have given me means more to me than I can express, and there are far too many to mention them all here, but they all hold a place in my heart.

I would like to give a special thanks to the following people:

My current employer Astorino, especially Louis Astorino, Dennis Astorino, and Bob Beckjord, who have been infinitely understanding, accommodating, and supportive throughout this process.

My previous employer IDDL, especially Tom Wilkinson and Mark Raby, who opened up the world of online education to me.

My friends, both from Blacksburg and Pittsburgh, who have let me bore them and even ignore them on so many occasions.

Miriam Guthrie and Charles Lattimer, for challenging me to be more than "all talk".

Thanks to my committee for planting the idea in my head, teaching me what I needed to know to make it happen, and being extremely patient with me.

Thank you to my family, especially my parents Thomas and Susan and my sister Katherine and brother-in-law Austin, for always pushing me forward, whether I wanted them to or not.

Finally, a special thanks to Liz Schmerling, for her inspirational spirit.

TABLE OF CONTENTS

FIGURE LIST

1.0 INTRODUCTION

The purpose of this master's thesis is to provide a system for the development and delivery of interactive 3D content as a component of web-based rich-media presentations, providing developers with a simple method for delivering visuospatial elements in online instruction.

Instructional content delivery via the internet is fast becoming a common experience in educational settings, from simple transfer methods such as e-mail and websites to more advanced methods such as streaming multimedia and audio-graphic conferencing systems. Many students of today are also quite familiar with interactive three-dimensional virtual environments rendered in real-time, through video games played on computers and game consoles. While solutions exist that offer similar virtual environments through web based interfaces, development of these solutions tends to be time-consuming and difficult, requiring high levels of 3D and programming expertise. Subsequently, development of 3D content for online instruction remains outside of the capabilities of most educational developers.

This project attempts to bridge that gap by bringing 3D capabilities to rich-media presentations and simplifying the development, connecting a familiar authoring environment to a commonly available real-time 3D solution.

Dubbed "XDRIVE", an acronym for XML-Driven Real-time Interactive Virtual Environments, this engine was designed to lower the expertise requirements that currently limit the amount of real-time 3D instructional presentations being delivered online. To accomplish this, an instructional language was built around the key actions that apply to elements within a 3D environment, and the systems for coordinating and executing the playback of those actions were built into the engine. The engine holds no predefined concept of the presentation that it will display, instead the presentation and connections to its elements are entirely defined by instructions imported at the beginning of playback from a file external to the player. To create XDRIVE presentations, a developer only needs to have a 3D model formatted in the Shockwave3D format (W3D), and a text editor to author the XML based instruction language. To view XDRIVE presentations, a user only needs to have the Shockwave plug-in installed on the client system.

Originally conceived as a method to enhance architectural walkthrough animations, the open nature of the XDRIVE engine extends its potential use to any area of information display and communication that may be improved through the incorporation of real-time 3D environments. In this manner, the system is meant to assist developers in bringing advanced web technologies into their creations. It is also possible that the development of presentations could be automated and even customized to individual instances by dynamically generating the instruction file via external systems.

At present, the actions available in XDRIVE's instruction language focus on control over elements within the 3D component, and the navigation and interaction of the user; the internal functions are primarily focused on the support of these functions and the stability of the presentation. In the future, the system could be further developed to accommodate additional functionality, support new instructions, and allow for the display and control of more media types. It could also be enhanced to send and receive communications with server technologies or with other client machines, facilitating multi-user environments.

1.1 PROJECT DESCRIPTION

The XDRIVE engine is a runtime solution for the coordination and display of multimedia presentations that feature three-dimensional content through a web-browser. This 3D content is rendered in realtime, which facilitates user-defined navigation and interaction with objects contained within the 3D virtual environment. These presentations can run independently, or they can be synchronized with audio and video files.

As web browsers interpret HTML formatted files, XDRIVE presentations are authored in and interpreted from XML formatted files, which are loaded and interpreted by the engine to display the defined content. Just as web browsers can load and display external files as guided and linked by the HTML tags, XDRIVE presentations rely on links to external files that are imported and displayed as guided by the XML tags.

Developed using Macromedia Director MX – a multimedia development software package - the XDRIVE engine itself is a Shockwave file that is embedded in a web page. Shockwave, a format whose browser plug-in is free to install and is loaded on a variety of systems, allows for the coordination of multiple media and data types, and features a powerful set of tools for the use of 3D content through the Shockwave3D format.

XDRIVE is designed to open the functionality of web-based 3D to a wider audience – allowing for custom presentations to be authored without a prerequisite knowledge of complicated programming languages, and 3D scripting. The XDRIVE engine is a series of scripted systems that utilize and connect various components of Director, and provide additional capabilities above those that already exist.

1.2 INFLUENCES IN DEVELOPMENT

Over the past 10 to 15 years, there have been a number of applications and data languages that have made their contributions to the advancement of web-based 3D and multimedia, and have served as points of inspiration for this thesis. The following have provided me some of the the greatest influence during my development of the XDRIVE engine.

- **VRML:**

  In the mid-nineties, as the World Wide Web was beginning to undergo widespread use, VRML (Virtual Reality Markup Language) was developed as a method to display realtime 3D environments over the web. These files defined the entire 3D world, including all geometries, lighting, and textures. Soon afterwards, version 2 (dubbed VRML97) was released as an ISO specification, to add animation and interactivity events to the system. Though there were many clever attempts to integrate 3D content with web pages, VRML was complicated to author, there were few browser plug-ins available to view the files, and the applications that could display VRML content were slow and often featured navigation methods that were complex and unintuitive. Consequently, outside of academic and high-end computer graphics circles, VRML never truly achieved widespread acceptance.

- **Multiplayer Video Games:**

  As early as VRML appeared to the public, it was far from being the first system for networked 3D environments. Doom (1993) and Doom II (1994), written by id Software, had already been on the market for years when VRML was released. These video games were amongst the earliest "first-person shooter" games – a term referring games in which a player navigates a 3D world from the perspective view of the game character they control. The user interacts with other objects within that 3D world (usually by shooting at them, hence the first-person "shooter"). One of the featured playing methods of this game was a "multiplayer" option where 2-4 different players on different computers would play a cooperative or competitive game, coordinated over a standard network. These games were wildly popular, and numerous games similar to these have been released over the years. As the games developed, the keypad navigation of these games evolved to include mouse input, which was used for id's "Quake" (1995) as a method of controlling the player's orientation. The "first person shooter" is now one of the most common video game genres, and subsequently, one of the most encountered navigation methods for 3D environments.

- **SMIL:**

  One of the specifications released by the World Wide Web Consortium (W3C), SMIL (Synchronized Multimedia Integration Language) was designed for the simple authoring of "rich-media" or multimedia applications. This XML style language was written to create presentations with audiovisual components that could be coordinated and timed with other media objects, primarily images and text. These presentations could be authored using a simple text editor, and could be viewed over the web using commonly available applications and browser plug-ins, such as RealPlayer and Quicktime. Most important to note, none of the media elements used in a presentation were contained or generated within the SMIL file – rather, media elements resided outside of the SMIL file, referenced through links.

- **Shockwave 3D:**

  For years, Macromedia Director has been one of the leading applications for the development of multimedia applications. Designed to build raster-based animation applications for CD and kiosks, Director has evolved over the years to include support for multiple media formats and communication methods, as well as its own object-oriented scripting language – "Lingo". As internet browsers grew in popularity and began to add support for third-party application plug-ins, Director added the capability to produce applications that would run in web page through the use of a plug-in, dubbed "Shockwave". With the release of Director 8.5, Macromedia incorporated support and programming for real-time 3D content as a new component. This 3D component was named "Shockwave 3D", and represented a new method for opening the world of 3D programming to a new market of multimedia developers by simplifying the concepts and programming knowledge required to develop such applications, all the while allowing developers to incorporate the other features supported by Director as well. This move also greatly increased the potential distribution, as the Shockwave plug-in is one of the most widely installed third-party plug-ins for internet browsers. This ease of programming, coupled with the fact that the proper software is already installed on most computers, makes Shockwave3D the current predominant method for delivery and display of web-based 3D content.

- **X3D:**

  Recently, the X3D specification has been released by the ISO as the successor to the VRML standard. X3D (eXtensible 3D) allows for multiple encodings of the 3D data, including the old VRML format and a new XML based format, has new added capabilities for improved graphics, animation, and networking components, as well as newer geometric formulations, such as NURBS surfaces. X3D still relies on data contained primarily within the file itself, rather than connecting to external data. While there are some publicly available browsers for X3D, it is still in the early stages of development and has yet to find a widespread market, though it does represent the best current opportunity for a common open-source 3D language.

Through my use or awareness of these applications and languages, I have been influenced in the design the XDRIVE engine. My intention was to design an application that would fit what I see as a missing component in web-based 3D – namely a simple method to author a web-based 3D presentation.

My work as a web-developer for Virginia Tech's Institute for Distance and Distributed Learning provided a great deal of insight into the development and delivery of online educational components, and to perform exploration into new methods of utilizing technology to enhance course delivery. Much of my work centered on the development of audiovisual presentations intended for remote delivery. For this purpose, SMIL was used to allow presentations to be streamed efficiently to clients with a wide variety of connectivity capabilities. A user client system downloading a presentation would only have to download the images used for a slideshow, and then stream the audio of the lesson encoded at a quality level corresponding to what the client connection could handle. This method proved to be far more efficient than delivering video content – images only needed to download once to display, while video of a slideshow required a constant stream of video information. By removing this network requirement, more bandwidth was available for the audio component, resulting in an improvement of the quality of audio delivered in the system.

The use of SMIL as a course delivery method also served to simplify the authoring and editing process. Compressed video created centralized presentation files – all content (audio, images, captioning) was contained within one file. Development required specialized applications that were often expensive, and time to compress the files from the source material. Once compressed, editing the file was usually impossible – instead, edits needed to be applied to the source materials which then required compression once again. On the other hand, SMIL files allowed developers to author files using only a text editor, they required no compression, and making changes to the presentation were as simple as editing a text file, or replacing an image. Changes made to the presentations by editing the SMIL file were instantaneous – no re-compression was required, as long as it was not the audio component that needed to be changed.

Prior to my employment at IDDL, I had been employed by the College of Architecture to work on a Cybercore History Project where I was involved in the development of 3D models for a course dealing with the history of monastic architecture. One of the cathedrals modeled for this course had been rendered as an animated walkthrough, and a series of QTVR panoramic images. These renderings facilitated a greater overall understanding of the subject matter, and of the spatial relations of the cathedral. This traditional style of computer animation had some drawbacks – rendering an animation video required a large amount of time, and allowed for no exploration of the space apart from that single video perspective. There was also no method for interacting with the content.

It was a subsequent discussion on the merits of delivering educational content for architecture over the web that first sparked the idea for including 3D media to the multimedia presentations as a meaningful method of content delivery. The restrictions of traditional 3D animation led me to explore alternate methods to display real-time 3D content that would facilitate user navigation and interactivity, such as video game engines. One of the main benefits of this method is that by using a real-time solution, the model would have limitless display potential, and a smaller file size. By moving the model rendering tasks to the client system instead of delivering video files of animations rendered prior, the client system would only need to download the file of the model itself, and the user could guide the navigation through the system.

I settled on Director after its addition of Shockwave 3D. Having previously developed an SMIL authoring application using Director, I was familiar with the structure of the program, the scripting languages, and the concept of connecting the built-in components. With those components, it became easy to add other capabilities such as audio and video, and file importing.

Finally, taking a lesson from editing SMIL file, I decided to use a similar structure to defining presentations – all media exists solely outside of the presentation file, connected through links and controlled through commands. To facilitate simple authoring of presentations, I chose XML as the data format for the familiar syntax, the ease of editing a text file, and the already existent components in Director for the importation and parsing of XML data.

1.3 MACROMEDIA DIRECTOR TERMINOLOGY

Macromedia Director began as a raster-based animation application. This should not be confused with vector-based animation applications such as Macromedia Flash, though Director does support the display of Flash content.

Originally designed as an application for CD and kiosk applications, Director has evolved into a multimedia development platform complete with its own object oriented programming language named "Lingo".

Projects developed in Director are exported for use in one of two ways: as a self-contained executable application known as "Projector", or in the "Shockwave" file format which is included as content in a web-page through the use of the Shockwave plug-in or ActiveX control.

Director describes its parts using a nomenclature derived from stage and screen influences. Projects are referred to as "Movies", and the library of component objects for a project is known as the "Cast". These objects contained within the cast are referred to as "Members". The display screen for the movie is called the "Stage", and the timeline of events are managed through the use of the "Score", a graph of the objects that occupy the numbered channels during "Frames" of the animation. An object that is on the stage is referred to as a "Sprite", and only one sprite can occupy a channel at a time, but multiple sprites can be inhabited by instances of the same member.

Objects in Director have various attributes and properties that can be tested and set using Lingo "Scripts", collections of functions that are compiled and operate during runtime. Object properties are accessible using a dot syntax similar to most other object-oriented languages. For instance, the channel number of a sprite can be retrieved by testing the "spritenum" property of the sprite by setting a variable to equal "me.spritenum". This function will only work when it is part of a set of functions known as a "behavior script" that is attached to a sprite on the stage. These behaviors are only called by the object to which they are currently attached. Scripts of functions available at a global level are referred to as "Movie Scripts", and are available to all objects. Scripts can also be attached directly to cast members, though this is not used for the XDRIVE engine.

Additional functionality is also available through the use of "Xtras", components that further extend the capabilities of applications, which are similar in nature to Dynamic-Linked Libraries.

There are multiple types of variables supported by Director, the most common being strings, integers, and floats. List arrays are available, either as a linear list or as a "property list" which stores data with named indexes which are also be recognized as data object properties. Variables can be assigned "global" status, which makes their value accessible to all functions so long as the function's script contains a reference to the global variable.

For this document:

Director attributes and functions will be identified, and contained within quotation marks.
Bold type face indicates functions or attributes written as part of the XDRIVE system
Italic type face indicates global variables used as part of the XDRIVE system

2.0 ENGINE OVERVIEW

The XDRIVE system builds upon the architecture of Director to create a playback engine for presentations with 3D content that are interactive and navigable, and operate along a set timeline. Playback relies on imported media resources, commands and settings, as well as input from user devices - the keyboard and mouse. It coordinates and executes events synchronized with the timeline across all objects in the presentation both inside and outside of the 3D component, and provides a level of control over the objects within the 3D world itself. Director provides access and control over the various components used (3D resources, file imports, AV object playback, image displays…), but it is the task of the developer to construct the architecture that will connect these components in the desired fashion.

At the heart of the application is the consistent evaluation and updating of the current state of the presentation and elements contained within. Once playback of a presentation is underway, the "command loop" repeatedly executes, sending the necessary events to each of the various control systems prior to the display of each frame of the animation. The command loop serves as the controller for the timeline of a presentation, evaluating the time elapsed since the previous frame completed rendering and updating the current time position along the defined series of events in the presentation, as is by the internal clock of the client machine, or the current playback time of an external audiovisual element.

User navigation events are evaluated by a function called from the command loop that listens for user input from the keyboard and mouse, and if those input criteria are met, updates the camera system by applying an incremental transformation designated by the specific type of input.

Timed events are cataloged and stored within the event queue system, with each event holding a specific start time and duration that will be evaluated against the current time for execution, an event command, and command specific attributes. Before each successive redraw of the stage and contents of the 3D sprite, the command loop updates the current time, then issues commands to check for user navigation events, to update certain trigger and texture system, and finally to update the items in the event queue to reflect the current time.

This process is illustrated in Figure 2.1:

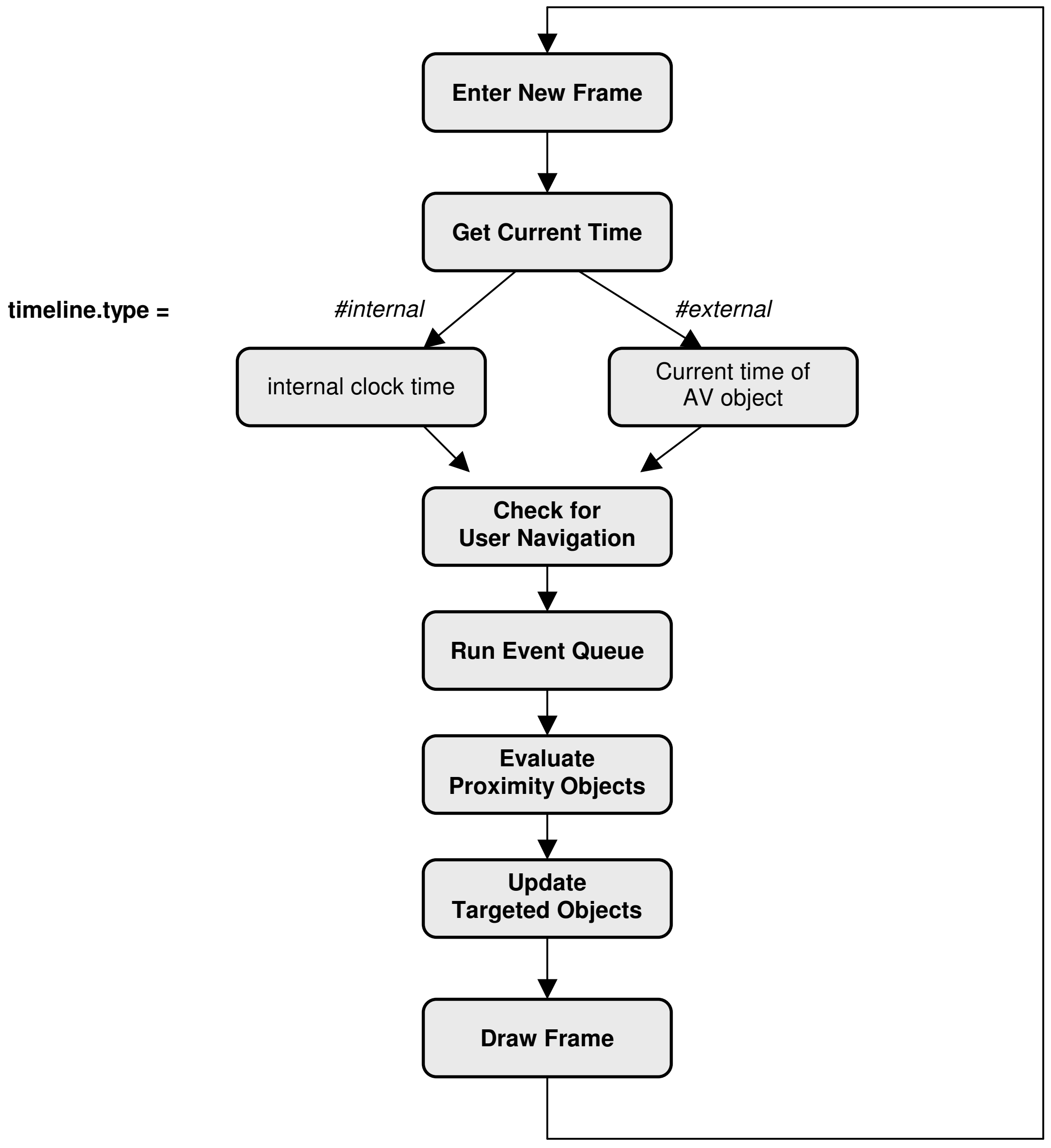


Figure 2.1: Command Loop Diagram

The event queue system checks all cataloged entries against the current time of playback for events that should begin, undergo, and complete their execution. Simple events – events that only have a start and end state with no intermediate values – only require one execution to complete. Animated events – events that require a series of incremental changes on each frame throughout the duration to traverse one state to another – require interpolation between the start and end states to determine intermediate values for the state, while the current time falls between the start time and end time of the event. Once the current time falls beyond the end time of the event, the event is executed to completion then removed from the system.

The event queue system is illustrated in Figure 2.2:

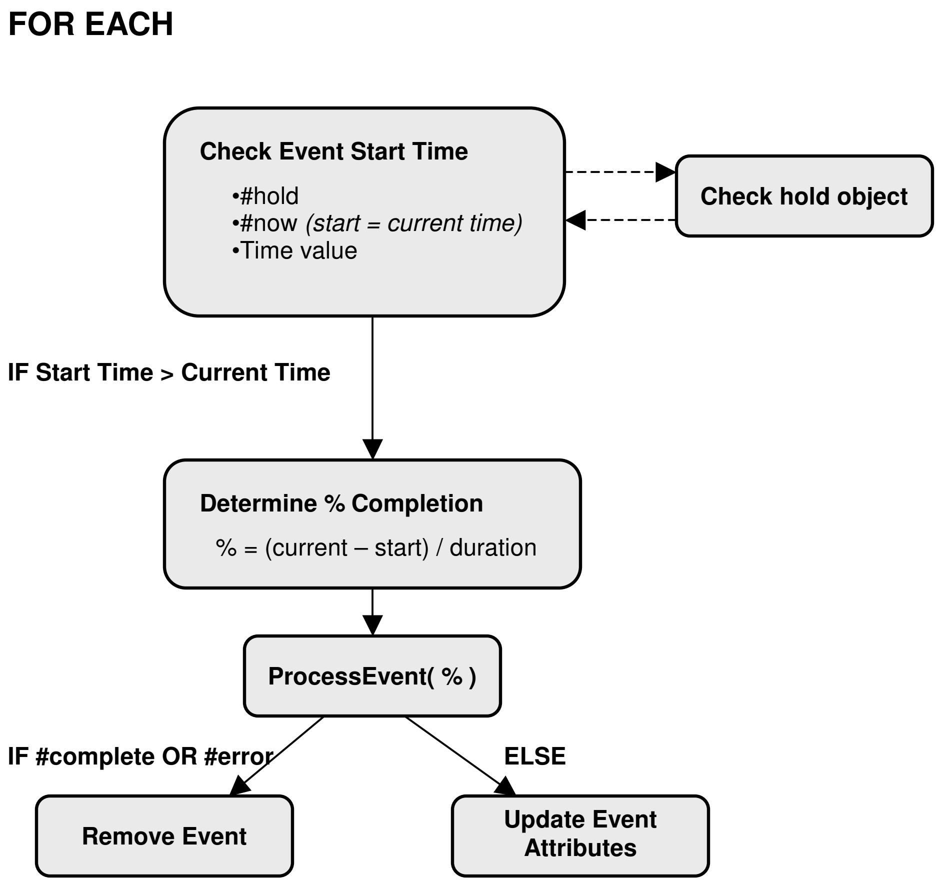


Figure 2.2: Event Queue System Diagram

2.1 ORDER OF EVENTS

To operate correctly, the timeline, navigation, and event queue systems require settings that are assigned prior to playback. The engine must initialize, load and parse the source file for the presentation, import and initialize the linked media, populate the cataloging systems, and then begin execution.

At startup, the application initializes the variables used by the system, and creates the default settings for those systems. The file path to the XML source file is determined and the file is loaded, and the setup tags are parsed and presentation settings are updated. The XML world and scene tags are cataloged. The external 3D resource file is loaded into the cast. Once completely loaded, the event queue and navigation systems are initialized, the stage layout is set, and the camera system is generated. The 3D world is added as a sprite to the stage, and XML containing the world presets for the presentation are parsed. The XML for the current scene is loaded and parsed, and the events are added into the event queue. Finally the timeline system is reset and initialized, and playback of the presentation begins. Once in playback, the system continues to loop on the same frame within the Score, calling the command loop function upon each instance of the frame.

This process is illustrated in figure 2.3:

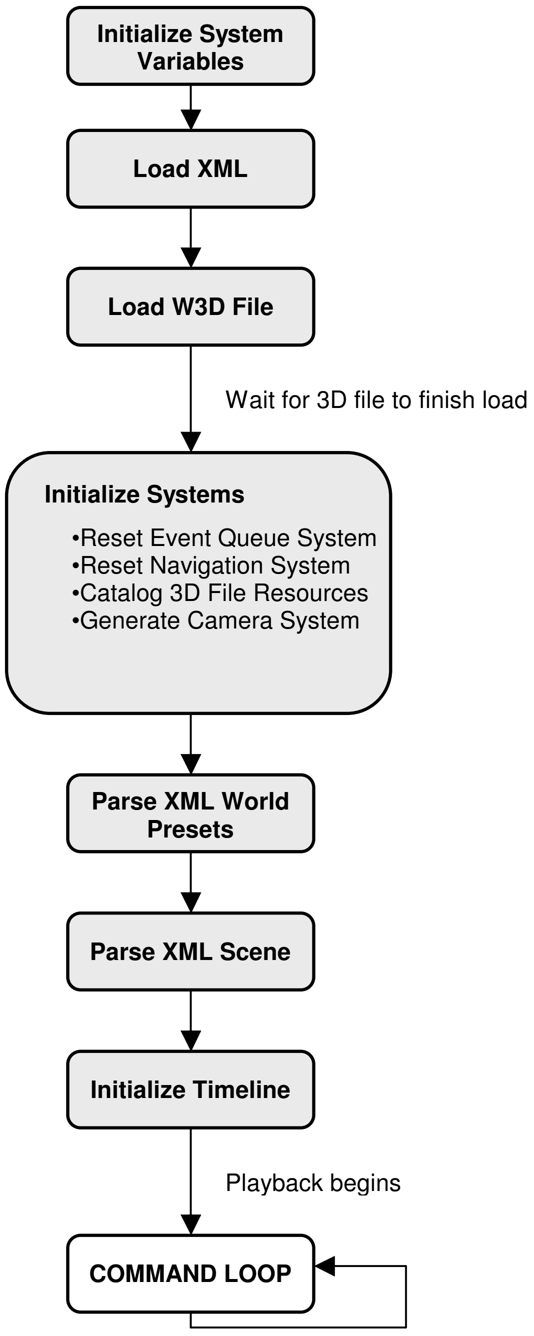


Figure 2.3: XDRIVE Order of Events

3.0 TIMELINE MANAGEMENT SYSTEM

To maintain a consistent animation, the XDRIVE engine employs a time management system which allows it to keep events within playback chronologically correct. This system, herein referred to as the "timeline" repeatedly evaluates the presentation's current time, serving as a command loop for the presentation.

Typically for animation, a frame rate of 24-30 frames per second (fps) is recommended. For this system, the timeline command loop is called from a behavior attached to the 3D sprite, evaluating upon each successive frame. The set frame rate of playback is 25 fps, or approx 40ms between frames, but because actual time between frames can vary, an independently measured time source is necessary to ensure synchronized playback. The timeline supports two sources of time readings – internal and external – at least one of which will be employed during playback.

The internal timeline interfaces with the playback computer's internal clock, which is measured in milliseconds. The current computer clock reading is then subtracted from the milliseconds measurement read at a prior checkpoint to determine the current playback time. This checkpoint is held by the global *loopstarthold*. For presentations employing only the independent timeline, the checkpoint is the clock reading at the start of playback.

- **current time = computer time – (computer time at loop start)**

(Figure 3.1)

The external timeline evaluates a current playback time by interfacing with a media Xtra - a built-in component that manages the playback of media components. In Director, the RealPlayer Xtra only evaluates a media component's playback time a few times per second, which resulted in a stuttered playback of animation events. To correct this, external media file time readings are only taken every 2 seconds, and the internal timeline is used to evaluate frames in between readings. At each external time reading, the *loopstarthold* value is updated to match the time. This allows for the timeline to accommodate interruptions in playback from networked files, such as stream buffering, and remain synchronized with the presentation.

The timeline is managed by functions contained in the movie script "Timeline system". These functions are: resetTimeline(), initializeTimeline(), commandLoopCall()

The timeline is referenced through the global *timeline*, which is defined by the **resetTimeline()** command. The *timeline* global is a property list containing the following:

#start: time (ms) [integer] – the start time for playback
#current: time (ms) [integer] – the current playback time
#type: #internal OR #external – the playback time source indicator
#navigation: TRUE or FALSE – determines whether the system will read navigation

queues
#position: TRUE or FALSE – meant for boundary, collision, & proximity events. NOT CURRENTLY ACTIVE
#attributes: [property list] – hold external timeline attributes

To set up the timeline, the **initializeTimeline()** command is called once the 3D sprite and presentation settings have been loaded. This command sends a **resetTimeline()** event, then sets the *timeline* attributes to match those from the global *currentscene*. If the timeline settings indicate an external media file will be used as the time source, a command is sent to initialize that media object as well.

Once the timeline has been initialized, the movie frame enters into a continual loop on itself. At each new frame, the 3D sprite send a **callCommandLoop()** event. This event performs the following tasks:

- Evaluate the current time from the internal or external time source
- Perform navigation events
- Animate the highlighting system
- Send the **DoQueuedEvents()** command
- Check for proximity trigger events
- Update object targets

4.0 EVENT QUEUE MANAGEMENT SYSTEM

The event queue system manages the operations of a presentation, ensuring proper chronological occurrence of the playback along the timeline. It serves as the central storage point for all commands issued during playback, queuing timed events, or interpolations of those events that have started but not yet completed, and removing events once they have fully executed. It also serves as a reference for other events that hold their execution until a specified event has completed.

The event queue system has five stages:

- Events are added to the queue
- Command loop (timeline) calls the queue system to evaluate.
- Current time is determined
- Events occurring during the current time are executed
- Completed events are verified and removed from the queue

Within the event queue, each event is stored as a property list, containing specific values required for the management of the system. The event structure variables are: name, command, starttime, duration, priority, holdtrigger and attributes.

#name: Contains the name for this specific event. Typically this name is assigned by the system, often including a randomly generated number to avoid duplicate names, but names can also be assigned by the user by providing a value for the "name" attribute for an event in the XML file. An event that is set to hold for a named event will continually check the queue for that name. When that named event no longer exists, the holding event will begin.

#command: Contains the event command name that should execute. When an event is determined to have occurred or be in the process of occurring, the queue determines which commands to send through a case-based comparison of this variable to the predetermined list.

#starttime: The time in milliseconds for which this event is to begin, with respect to the playback timeline. Once the value of the timeline's current time is greater than or equal to this time, the event is considered to have begun, and is evaluated for execution. There are also two special starttime values, **#now** and **#hold**. **#now** indicates that upon it's initial evaluation, the system will execute that event immediately, replacing the start time value with the timeline's current time. **#hold** indicates that this event is waiting for the completion of a named event before it's execution. Upon evaluation, the system checks the queue for an event with a **#name** variable matching that of the current event's **#holdtrigger** value. If that name is not found - it has either been removed from the list due to completion or cancellation, or did not exist there to begin with – then the event begins execution, replacing the start time with the timeline's current time.

#duration: The time in milliseconds for which this event will remain in execution. If an event has begun (the current time is past the **#starttime** value) the duration is evaluated to determine an end time for the event. If the time between the current time and the start time is less than the duration value, then the event is only partially complete, and a percentage complete value is used to interpolate the completion, where appropriate. If the current time is greater than or equal to the start time plus the duration, the duration is set to **#expired** and the event is executed to completion. Events are then typically removed from queue upon their completion.

#priority: sets the priority level of an event to one of three values: **#verify, #noverify,** or **#once**. When the queue system encounters a #priority value of #verify, it will not remove the event from the queue system until it has received positive feedback from the event processing system that the event has completed. A #noverify value indicates that the event should be removed from the system after it's duration has expired, disregarding feedback from the system as to the success or failure of the command. A value of #once indicates that the event should be removed upon the first successful completion of the command. (Feedback can also be a #cancel value, indicating an error with the system. Events that encounter a #cancel value are automatically removed from the queue.)

#holdtrigger: holds the name of an event for which this event is waiting for the completion of. Once that named event does not exist in the queue, this event will execute.

#attributes: these hold the attributes for the event which are specific to the event itself, and vary by command. Typically these attributes will include references to objects, as well as start and end values that will be interpolated upon execution. They may also sometimes contain the previous interpolation percentage, in order to determine elapsed time increments for interpolation of relative events.

The event queue system is comprised of four commands: **resetEventQueue(), addEventToQueue(), doQueuedEvents(),** and **processEvent().**

resetEventQueue() sets the global *eventQueue* to an empty list, clearing the queue of any artifacts or leftover events.

addEventToQueue() appends the *eventQueue* list with an event passed in through the command. This is the only method by which events are added into the queue.

DoQueuedEvents() evaluates all queued events against the current time, determined by the global *timeline*. This command is repeatedly called by the command loop running through the timeline system. After determining the current time of playback, it evaluates each event contained within *eventQueue* using the following method:

1. Check for special event start times:
    a. If the start time is #hold, check *eventQueue* for a named event with this name. If no event exists, set the start time to the current time.
    b. If the start time is #now, set the start time to the current time
2. Check the event start time against the current time

    a. If the current time is less than the start time, go to the next event
3. Check the event end time
    a. If the current time is greater than or equal to the event start time plus the duration, set the duration to **#expired**
4. Set the event completion percentage (decimal)
    a. If the event duration value is #expired then the percent complete is 1, or 100%
    b. If not, evaluate the percentage of completion by dividing the event's elapsed time (current time – start time) by the total time (duration)
5. Send the **processEvent()** command, with this event and the percentage completion.
6. Evaluate the response from the **processEvent()** command.
    a. If the response value is #cancel, remove this event from the queue
    b. If the response value is #true, and the duration is #expired, remove this event from the queue
    c. If the #priority value is #noverify, and the duration is #expired, remove this event from the queue

ProcessEvent() evaluates and updates the event passed to it. The command receives four variables – the event number (index of event within *eventQueue*), the event command name (the #command value), the event attributes (the #attributes value), and the event's percentage completion. A case-based comparison of the event command name determines which script to run. This script is command specific, and will run either a direct command or an interpolation command (using the percentage complete). Feedback from this command will determine whether to send back #cancel or #true feedback response to the **doQueuedEvents()** command, or in some cases, whether to update the attributes of this event in *eventQueue*, as indexed by the event number.

5.0 INTERPOLATION

"The foundation of most animation is the interpolation of values." [Parent, p. 63]

Interpolation refers to the process of using mathematical functions to estimate values between known data points. For an object at Point A to appear to move to Point B over the course of 10 seconds at 25 frames per second, the computer must display 250 images of that object, each one positioned incrementally further along the imaginary line between A and B. Rather than storing 250 discrete position points for the object, interpolation functions let the computer calculate the appropriate position of the object between just two position points – A and B. However, interpolation's role in animation is not confined simply to linear motion. It can be applied to any animated event that calls for a gradual change in value.

The XDRIVE system uses interpolation for a number of apparent purposes - motion, rotation, color changes, opacity change – as well as some processes that are not so apparent, such as path direction and distance estimation. The most common interpolation type is linear interpolation – a shortest-distance approach to finding an internal value between to data points using a percentage approach, but some interpolation functions can be complex, such as the Catmull-Rom spline, used for creating smooth animation paths through a series of position points.

5.1 COLOR INTERPOLATION

Color values are expressed in Director as RGB values, which are comprised of intensity values for the Red, Green, and Blue channels (the primary colors of light). Each channel has 256 potential values, indicated as 0 for no intensity, and 255 for full intensity. Thus a pixel with an rgb value of (0,0,0) will appear black, while a pixel with a value of (255,255,255) will appear white, and a pixel with a value of (127,127,127) appears as 50% gray. The number 256 is not arbitrary – it is the number of possible values in 8 bits, or 2^8. Three channels, each with 8-bits are commonly referred to as 24-bit color, meaning 2^8 (red) * 2^8 (green) * 2^8 (blue) = 2^{8+8+8} = 2^{24} possible values, about 16.7 million possible colors.

Because Director recognizes the three intensity channels and handles them mathematically in the same manner as it handles vectors, it is possible to apply scalar values equally to all three values, and to add & subtract values. To interpolate color in the XDRIVE system, the **interpolateColor()** command is used. Three variables are received, the starting color, the desired end color, and the percentage between the two that is to be linearly interpolated. The following functions are used to interpolate the color:

- **Color 1 = Start Color * (1 – Percentage)**
- **Color 2 = End Color * Percentage**
- **Interpolated Color = Color 1 + Color 2**

(Figure 5.1)

When the percentage is low, the start color will be the dominant color as the end color is multiplied by the smaller percentage. As the percentage approaches full value (1.0), the value of the end color will be the dominant color, as the value of the start color is negated by multiplying it with a value near 0.

5.2 OPACITY INTERPOLATION

XDRIVE uses linear interpolation for calculating the appropriate transparency level with which to display opacity and fade animations. The formula is simply:

- *If the start value is greater than the end value (fade out)*
  Interpolated Value = Start Value – ((Start Value – End Value) * Percentage)
- *Else if the start value is less than the end value (fade in)*
  Interpolated Value = Start Value + ((End Value – Start Value) * Percentage)

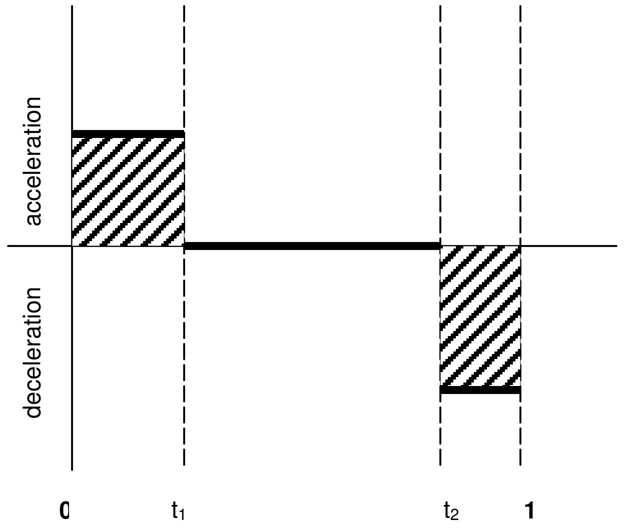
(Figure 5.2)

5.3 MOTION AND ROTATION INTERPOLATION

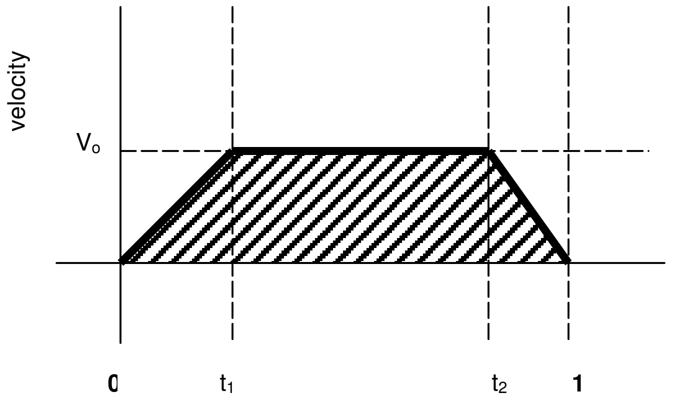
For motion and rotation animations, linear interpolation is again employed to determine the position along a path between keypoints, or the angle of rotation between start and end angles. In this case, it is the percentage of completion that is being passed to the appropriate interpolation commands. Due to the linear nature of time, the end value (1.0) will always be greater than the start value (0.0), so the percentage itself usually needs no interpolation itself. For relative motions and rotations, that is, animations whose end values are not discrete positions or directions but rather are relative to their start value, (such as an object rotating 360 degrees about the z-axis regardless of any other rotation events) the motion or rotations are set incrementally upon each execution. The values passed execute a percentage of that relative animation, defined as the value of the current percent complete subtracted by the percent complete upon the previous execution which is stored within the event's attributes.

5.4 ACCELERATION AND DECELERATION

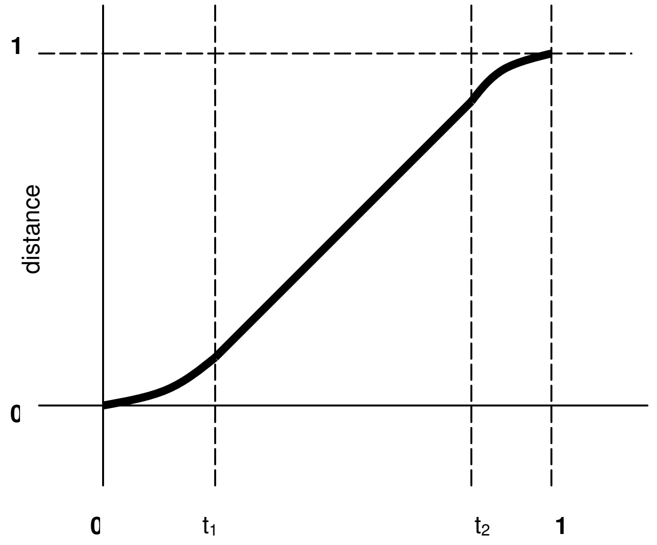
Linear interpolation of percentages of completion will result in a constant speed in animated events from their start to finish. Objects at rest instantaneously achieving their maximum speed is an unnatural phenomenon, and can create an unrealistic appearance. It is therefore desirable to provide a method to allow objects to accelerate to and decelerate. These gradual increases and decreases in speed where motion or rotation events take a defined period of time to move from one velocity to another, are referred to as "ease-in" and "ease-out", in reference to the curved appearance of portions within a distance graph where acceleration or deceleration occurs. Though used most often for animated motion, this approach can be incorporated for any event interpolation that would otherwise be linear. For ease of explanation, the following descriptions use normalized "distance" (min value: 0.0, max value: 1.0) for motion events, but should be thought of as a percentage of completion for animated events.



(Figure 5.3 – Acceleration Curve)



(Figure 5.4 – Velocity Curve)



(Figure 5.5 – Distance Curve)

Figure 5.3 illustrates the ease-in / ease-out approach for an object that starts at rest at time 0.0, accelerates constantly until a certain time (t_1). Later, (t_2), the object decelerates until time 1.0. For the object to begin and end at the same velocity (in this case at rest, or 0.0), the times of acceleration (t_1) and deceleration ($1.0 - t_2$) do not need to be the same, but the areas (a_1) and (a_2) must be equal.

Figure 5.4 graphs the velocity over time resulting from the acceleration / deceleration shown in Figure 5.3. The velocity increases linearly between time 0.0 and t_1, and decreases linearly between time t_2 and 1.0. The maximum velocity (V_o) remains constant when no acceleration or deceleration is applied. The area under the velocity curve is equal to the distance traveled over that time.

The maximum velocity (V_o) can be expressed as:

- **$(V_o) = 2.0 / (t_2 - t_1 + 1.0)$** [Parent, pg 91] (Figure 5.6)

Figure 5.5 graphs the distance traveled over time resulting from the velocity curve shown in Figure 5.4. The distance has been normalized (values between 0.0 and 1.0). This decimal percentage of the total distance can also be considered as percentage completion of an event. Note that during the periods of acceleration and deceleration, the distance line is curved – these are often referred to as "ease-in" and "ease-out" curves.

5.5 FINDING DISTANCE BY TIME

The function **FindDistanceByTime()** is used to calculate this percentage of completion along a normalized distance curve. The three variables are:

- percentage of event duration (calculated as [event time elapsed / duration])
- the percent of time accelerating (calculated as [acceleration time / duration])
- the percent of time deceleration (calculated as [deceleration time / duration]).

The return value is the percentage of completion, between 0.0 and 1.0.

To calculate a distance (d) at a time (t), the following functions are used:

- **When:**
    - $0.0 < t < t_1$: $d = V_o * (t^2 / (2 * t_1)$
    - $t_1 \leq t \leq t_2$: $d = [V_o * (t_1 / 2)] + [V_o * (t - t_1)]$
    - $t_2 < t < 1.0$: $d = [V_o * (t_1 / 2)] + [V_o * (t_2 - t_1)] +$
    $[[V_o - [[V_o * (t - t_2) / (1 - t_2)] / 2]] * (t - t_2)]$

[Parent, pp. 91-92] (Figure 5.7)

(We do not need to calculate d when t is at 0 or 1. Since we are calculating normalized distance over normalized time, by definition the value of d at 0 is 0, and d at 1 is 1.) Finding Time By Distance:

5.6 FINDING TIME BY DISTANCE

There are a number of motion types that can be applied to objects or the camera. Often, a motion animation will be comprised of more than just one event, such as moving through a series of linear events, or animating the camera along a spline path through a series of checkpoints. This sequence of motions will still have one start time, duration, acceleration setting, and deceleration setting to define the entire event. In this case, the total animation event is actually comprised of a series of smaller motion events that are calculated by the computer during playback. To maintain a smooth animation across these individual events, it becomes necessary to pre-calculate the completion of the sequence over time as one continuous event, then use that set of completion values to assign start and end times to the individual events.

To calculate the times for events in a sequence, we can work backwards to *t*, using known values of *d* (normalized distance).

First it is necessary to determine the overall distance traveled by the object during the sequence, which is accomplished by determining the distance traveled in each individual event and taking the sum of the results. Since the overall distance is the sum of distance traveled in each individual event, dividing an individual event's distance by the overall distance will provide you with the decimal percentage that event contributes to the overall normalized distance.

- overall distance (d_o) = distance 1 (d_1) + distance 2 (d_2) + … distance n (d_n)
- normalized distance = (d_1 / d_o) + (d_2 / d_o) + … + (d_n / d_o)

(Figure 5.8)

Applying the same concept to the time value, divide the acceleration and deceleration times by the overall sequence duration to arrive at the normalized time. These normalized values are important because they decide which function we will use to determine the distance.

- normalized acceleration time (t_1) = acceleration time / duration
- normalized deceleration time (t_2) = (duration – deceleration time) / duration

(Figure 5.9)

Once the acceleration and deceleration have been determined, the maximum velocity (V_o) is calculated:

- $V_o = 2.0 / (t_2 - t_1 + 1.0)$ (Figure 5.10)

Understanding that distance is equal to the area underneath the curve in the graph of velocity over time, we determine three distances: the distance traveled during acceleration (d_1), the distance traveled while at maximum velocity (d_2), the distance traveled during deceleration (d_3).

- $d_1 = ½\, t_1 * V_o$
- $d_2 = (t_2 – t_1) * V_o$
- $d_3 = ½ (1 – t_2) * V_o$ (Figure 5.11)

Now that we have the distance values defining the acceleration and deceleration distances, the following formulas to calculate time (t) at a certain distance (dt).

If d_t is less than d_1, this distance occurs during the acceleration, before t_1. Since $d_t = V_o * (t^2 / (2 * t_1)$:

- $t = \sqrt{} [(d_t / V_o) * (2.0 * t_1)]$ (Figure 5.12)

If d_t is greater than d_1, but less than d_2, this distance occurs during the period of no velocity change, between t_1 and t_2. Since $d_t = [V_o * (t_1 / 2)] + [V_o * (t – t_1)]$:

- $t = (d_t / V_o) + (t_1 / 2)$ (Figure 5.13)

If dt is greater than d_2, this distance occurs during the period of deceleration, after t_2. Since d_3 is the area of the triangle defined by the velocity curve during deceleration, we can calculate the time t by using d_t to determine the distance not yet elapsed (d_x), then using dx to determine the time which has not yet elapsed. The final formula appears as:

- $t = 1 – [(1 – t_2) * \sqrt{} [(d – d_t) / d_3]$ (Figure 5.14)

5.7 SPEEDCHARTS

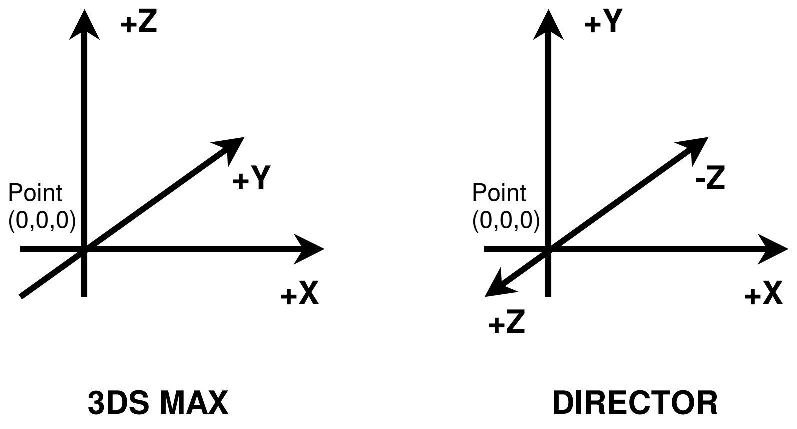
Once normalized times have been determined using normalized distance, the individual events within a motion sequence are assigned the appropriate start times and durations. However, it is possible that the start or end times of individual events may occur during acceleration / deceleration, spanning that acceleration and/or deceleration across multiple events. To manage the proper completion of such events, a speedchart is used for each event to manage the overall progress. During the pre-calculation of motion events, XDRIVE determines whether or not an individual motion event is subject to an acceleration / deceleration. If so, a chart of values is calculated for the percentage of completion of the individual motion event at increments of 5% of the event duration. This chart is then stored by the event, and upon execution, the percentage of motion completion is calculated through linear interpolation of the two values at the event times bracketing the current elapsed event time. If an individual motion event has no acceleration or deceleration applied to it, the speedchart value is stored as **#constant**, indicating that the percentage completed directly correlates to the normalized elapsed event time.

6.0 WORLD SETTINGS

The <worldsettings></worldsettings> tags provide the ability to define or override certain settings that apply to the overall 3D component of the presentation. The background color can be set, as can the overall orientation (position, rotation and scale) of objects contained within the 3D world, and the initial camera position. The camera headlight can be enabled or disabled, and the beam type (directional or spot) and color of the light can be set. The color used by the highlighting system can also be set, as can the duration of the highlight's pulse time.

6.1 WORLD ORIENTATION

In most 3D modeling programs, such as Maya and 3DStudioMax, the X and Y axis define a horizontal plane – the X axis corresponds to left and right, the Y axis to front and back – while the Z axis corresponds to vertical depth, or up and down. This is not the case with Director, which treats the Y axis as the default "up" for all objects, as shown in Figure 6.1.



(Figure 6.1 – Diagram of Default World Orientations)

This discrepancy between the default orientations of typical systems and Director resulted in the imported 3D world to be rotated on it's side when viewed with the camera object created during the program setup, essentially point the camera what would normally be considered "down". While it would have been possible to rotate the camera object, any new objects created would also all have their point-up orientation (a vector hinting at the up direction for an object), so my solution was to rotate the entire world instead. All XYX values input into the system from the XML document are re-interpreted to correspond to the new X & Y orientation using the following rules:

- x-value: no change
- y-value: -(z-value)
- z-value: y-value

Upon importing a 3D world, all objects within the world are grouped, rotated –90 degrees about the X axis through point (0,0,0), then ungrouped. This rotation can be altered by the author through the use of the <rotation> setting of the <worldsettings> tag in the XML document to create a desired rotation. The <position> setting allows the author to alter the relative initial position of the world, and the <scale> setting lets the author alter the relative initial scale of the world. The default rotation is X=-90, the default position is no position transformation – (0,0,0), and the default scale is no scale transformation – (1,1,1).

6.2 BACKGROUND LIGHTING

The 3D sprite has an attribute "bgcolor", which allows for definition of the color of pixels in the "background" – those appears behind all models in the scene. This value defaults to black (R:0, G:0: B:0), but can be overridden using the <worldsettings> tag. Functions are also included for animating background color changes, using interpolation between start and end color values to interpolate the change over a set duration.

6.3 WORLD LIGHTING

3D sprites in Director by default have directional lighting, and ambient settings for default shaders. This directional lighting has been disabled (set to "None") for XDRIVE presentations. All lighting is imported in the W3D file, with the exception of the camera headlight.

7.0 CAMERA SYSTEM

The "camera" represents the user's virtual position and orientation within the 3D world. It is this position and orientation that will be used by the system to render a perspective view of the world space to the 3D sprite on the screen. Typically, objects within the 3D world have 6 degrees of freedom:

- motion along the x-axis
- motion along the y-axis
- motion along the z-axis
- rotation about the x-axis
- rotation about the y-axis
- rotation about the z-axis

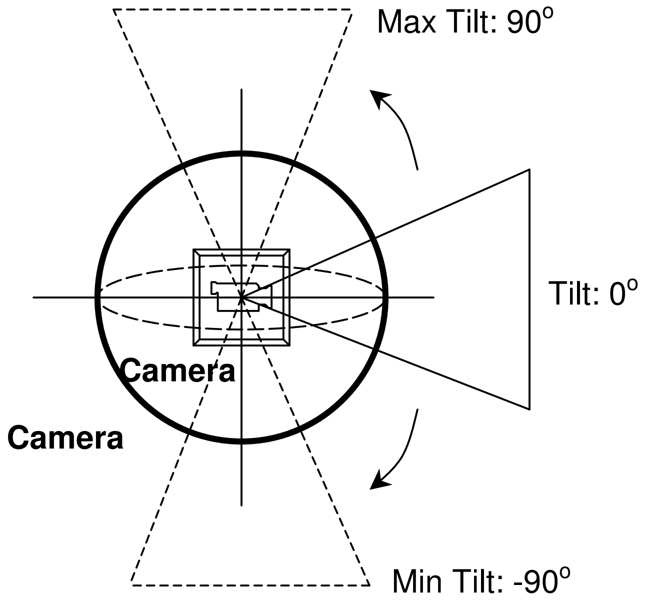
But for the camera, some of those motions and rotations are restricted or unavailable, for ease in navigation. Camera control in XDRIVE was designed to operate like a tripod, a method similar to many popular video games experienced through a first person perspective. This method of visualization restricts rotation of the user's orientation to "pan" the camera left and right (rotation about the z-axis), and tilt the camera up and down, (rotation around the user's relative X-axis). While users can pan indefinitely, their tilt is limited – they can tilt up and down only so far as facing straight up and down respectively. The maximum tilt angle is 90 degrees above or below the ground plane.

The camera's motion within the space is unrestricted by default, but can be set to limit user's motion to an XY plane by restricting Z-axis motion.

7.1 THE CAMERA OBJECT

The camera object in XDRIVE consists of two objects – the camera itself, and a model. The model is a small box whose visibility has been turned off. The camera is centered within the box, which is designated as the model's "parent", so any transformations affecting the box model will have an equal effect on the camera. A resource to the model is stored as a global variable in the system – *cameraobject* – a reference used by other commands to set or retrieve positions and orientations for the camera. This reference can also be employed for use in future developments of boundary and collision detection systems.

Camera rotation is restricted by only allowing two types – pan and tilt rotations. Pan rotations are achieved by rotating *cameraobject*, the parent model of the camera object, about the Z-axis. Tilt rotations are achieved by rotating the camera about the X-axis of the parent model. Tilt rotations are restricted to a minimum value of –90 degrees and a maximum value of 90 degrees. If a command sends an angle less than –90 or greater than 90, the tilt is set to the minimum or maximum rotation and the commanded angle is ignored.



(Figure 7.1 – Diagram of the Camera System)

7.2 NAVIGATION

There are two methods for controlling the camera's position and orientation within the 3D world:

- **System Control:** The camera's position and orientation is controlled through the use of events called by the timeline.
- **User Controlled:** The camera's position and orientation is controlled through input from the user in the form of key presses and mouse positioning.

These two control methods allow for three types of navigation:

- **Guided Navigation:** The system moves and rotates the camera to pre-set positions over a designated time. These changes in position and orientation are interpolated upon the execution of each frame.
- **Free Navigation:** The user moves and rotates the camera using the mouse & keyboard. These changes in position and rotation are incremental and occur upon the execution of each successive frame. Speeds for motion (in units) and rotation (in degrees) are preset into the system as incremental values.
- **Mixed:** Since the system always retains ultimate control of the camera, timeline controlled events are available to be applied to the camera at any time. Free navigation can be either enabled or disabled within the system. When disabled, user input from the keyboard and mouse are ignored.

User controlled navigational settings are contained in the system global *nav_settings*, a property list of the attributes pertaining to various aspects of the navigation control.

- **#type** indicates to the system what type of user-controlled navigation is to be implemented. Currently "free" is the only valid option, but open for development of other types, such as using the mouse to rotate an object, or rotate the camera about an object or position.

- **#mouse** indicate to the system whether mouse input is enabled, and contains the mouse values for the "speed" of mouse rotations (rotation angle increments) and whether or not to "inverse" mouse tilt rotation. It also defines the "buffer zone" for the mouse, an area over the 3D object within which the mouse can be positioned without resulting in a motion or rotation of the camera.

- **#keymap** defines the keys which when pressed will result in motion or rotation events.

- **#speed** defines the incremental distance to travel or angle to rotate for keyboard commands.

7.2.1 KEYBOARD BASED USER INPUT

There are 9 possible key events:

- move forward
- move backward
- move left
- move right
- move up
- move down
- rotate left
- rotate right
- rotate up
- rotate down
- toggle mouse-look availability

The keymap is a property list containing ASCII character numbers that associate a particular key with a particular event. Upon each execution of a frame during playback, if user-based navigation is enabled, the system checks the keymap for each possible motion or rotation event-key to see if that key is currently pressed. If it finds the key is pressed, that event is executed by rotating or moving the camera in the specified direction by an increment defined in *nav_settings.speed.* For the two toggle commands, the key is only evaluated during the initial "keyDown" event, since those only need to occur one time per press. Since the motion and rotation events test for a key being currently pressed, holding the key will result in that event repeating as long as the key is held. This also allows for multiple key-events to execute simultaneously.

7.2.2 MOUSE BASED USER INPUT

User input through the mouse is received and interpreted by behavior scripts attached to the 3D sprite. This allows the scripts to only analyze mouse events when the mouse the sprite, saving the system from expending unnecessary calculations when the cursor position is outside of the effective area.

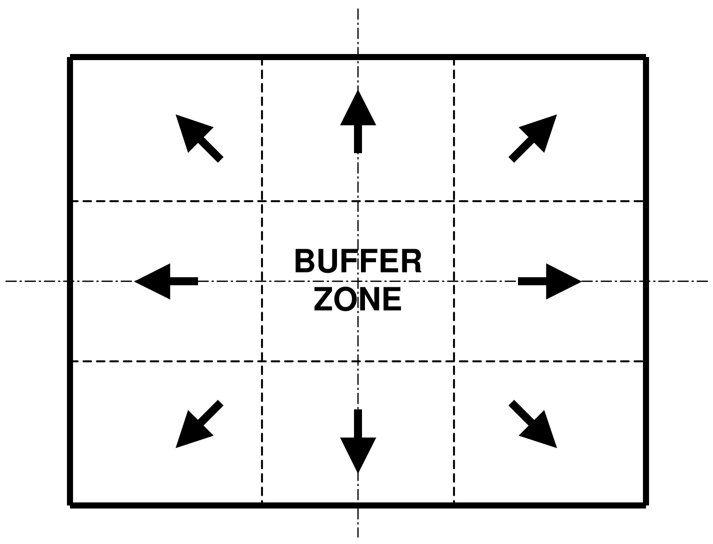
There are two types of mouse input handled by the 3D sprite:

- Mouse Look: When enabled and activated, the cursor position is interpreted relative to the center point of the sprite, when the mouse is over that sprite. The distance and direction is used to rotate the camera in the direction of the mouse, at a speed that is proportional to the distance from the center point.
- Mouse Click: When the mouse button is depressed while the cursor position is over the 3D sprite, that position is evaluated to find what models appear at that position in the display. Found models are checked to see if the top visible model – the model that was "clicked" – has a click trigger event associated with it. If a click trigger is found, that trigger is executed.

There are three functions within the behavior scripts for the 3D sprite dedicated to mouse events. The first script executes upon the creation of the sprite itself, and adds properties to the sprite that will hold attributes related to mouse navigation settings and to the sprite itself.

The second script is a "mouseWithin" script, that executes on every frame when the mouse is positioned over the 3D sprite. If this event executes, it first checks to make sure that mouse-look navigation is enabled – this option can be set in the <navigation> preferences in the XML file – and if it is active – pressing the mouse-look toggle key (default "m") will turn the enabled mouse look on / off. If both of these conditions are met, the mouse position is recorded. It is first checked against the buffer-zone, a rectangle within the sprite, proportional to the sprite by a percentage defined in the <navigation> tag. This percentage value is converted to a screen area during the navigation setup functions, and is stored by the sprite as a "rectangle" object. If the mouse is within this object, no motion occurs. It is then evaluated for its position outside of that rectangle for the pan and tilt rotations that should be send to the camera. These rotations are percentages of the incremental angle set by the "mouserotate" attribute in the <speed> tag, calculated as the percentage distance between the buffer zone and the edge of the sprite. Figure 7.2 illustrates the 9 areas created within the mouse-look areas.

Finally, a "mouseDown" event simply waits for the mouse to click on the 3D sprite. It takes the 2D position of the mouse, and sends that information to a script to check for click triggers.



(Figure 7.2 – Diagram of Mouse-Look Zones)

7.3 CAMERA FADE

Director supports background images and image overlays for 3D members. These are attributes attached to the camera, and those images can be set to have a "blend" or opacity value. The XDRIVE system uses this to create the appearance of the camera "fading" in and out, by overlaying a solid color over the camera. The blend value of the overlay is then interpolated over a set time duration, allowing the model to gradually appear, giving the appearance that the camera is fading in. The reverse situation is handled by interpolating the opacity in the opposite direction.

7.4 CAMERA HEADLIGHT

Since there is no way to predict the lighting that will exist in models imported into the system, a "headlight" function is included. When a scene has the camera headlight enabled – accomplished through the <headlight> tag within the <worldsettings> - the system creates either a spot or directional light of a specified color (type and color are assigned in the <headlight> tag). This light is then set to the same position and orientation as the camera, and parented to the camera. Now any camera transformation will equally affect the light as well, essentially fastening it to the camera as a headlight.

8.0 OBJECTS

There are four types of objects within Director's 3D member environment:

- Model
- Light
- Camera
- Group

Models are the visible geometries within the 3D environment, defined by a series of points connected by edges that form faces that are in turn rendered by Director. Models can be imported, or can be created and defined within the system. The XDRIVE system only supports models that are imported in the W3D file – no code has been written to create models from XML data, or import models from a W3D library, though this functionality may be added in the future.

Lights are used to illuminate models within the 3D world. They have a color, a direction, and an intensity. There are four types of lights which indicate the way in which light emanates from the point of the light source. "Ambient light" evenly illuminates all faces of models within the 3D world, regardless of position or orientation. "Directional light" casts light across faces in a single direction from the source point. "Spot light" casts light emanating from the source point in a cone whose angle can be set. "Point light" is a light that emanates from the source point in all directions. The XDRIVE system currently only supports lights that are imported in the W3D file.

Cameras define the viewpoint of a perspective into the world. They cannot be viewed within the 3D world, as they have no geometries. Instead, they represent the position and direction to be rendered for the 3D display. While Director supports multiple cameras within a scene, and the 3D member's view source can be switched between cameras within a scene, XDRIVE currently only supports one camera, the one created by the system during startup.

Groups are primarily collections of models, though other objects can be included in these collections as well – lights, cameras, and other groups. Groups allow all objects contained within the collection to be controlled as one object. XDRIVE supports the creation and destruction of groups.

All of these objects share a similarity in that they all have a position and orientation, and each instance of an object has a name attribute. By using this name as a reference to the object resource, the system can quickly retrieve or set the position and direction of any object currently within the world. In Director, sending a command to a named object that does not exist will result in a script error, and the program will stop. To prevent this from happening during playback, cataloging and verification scripts have been put in place for each type of object, ensuring that an object of that name exists somewhere in the system before any commands are issued using that name.

8.1 MODELS

Model type objects are the most commonly referenced by the system, as they comprise the content of a 3D world. Once Director has loaded the 3D world, and before the prescene events are executed, the **catalogModelNames()** function is called, which runs through every model in the 3D world and adds the name of each model to the global *current_model_list*. When models are called by name by a function, that name is first checked by the **verifyModel()** function. This function takes the model name as a variable, and looks for any instance of that string contained within the *current_model_list*. If it finds one, it returns a value of TRUE or 1, otherwise it returns FALSE or 0. This way, if a command calls a model that is not recognized as being in the system, the body of the command is not executed when the FALSE value is returned, preventing an invalid model from being referenced and crashing the system.

The "Model / Group Interaction Scripts" also provide functions for retrieving position, rotation, and scale properties of a model, as well as setting them. These are:

- **getModelPosition()** – returns the current position of a named model as a vector
- **getModelRotation()** – returns the current rotation of a named model as a vector
- **getModelScale()** – returns the current scale of a named model as a vector
- **setModelPosition()** – transforms a named model's position to the defined world coordinate vector
- **setModelWorldRotation()** – transforms a named model by rotating it around a defined axis by a defined angle, with respects to the world coordinates.
- **setModelSelfRotation()** – transforms a named model by rotating it around a defined axis by a defined angle, with respect to the model's self-coordinates.

8.1.1 POINTING

Every object has a direction that it is considered to be facing, defined by Director using the object's "pointAtOrientation", a list of two vectors; the first represents the direction – with respect to the object – that the object is "facing". The second represents the direction that is considered "up" for the object. When Director receives a "pointAt" command, it will rotate the specified object so that the "facing" direction is oriented towards the defined position, and will then rotate the object so that the "up" vector is pointing towards the same direction as the world's "up" direction. The model pointing scripts are:

- **getModelDirection()** – Retrieves the world coordinate direction of the "pointAt" orientation of a model. This direction is calculated by rotating the "pointAt" orientation (a direction relative to the model itself) about the axis and angle of the model's current world rotation. This is used primarily in for interpolation of animated rotations of a model as it moves to face a world position or orientation.
- **PointModelAtPosition()** – Sends a "pointAt" command to a model, facing the model's "pointAt" orientation towards that position.

- **PointModelAtDirection()** – Sends a “pointAt” command to a model, facing a position calculated as the current position of the model plus the directional vector.

It is possible to re-define the “pointAt” orientation of a model, which is useful when that model’s default orientation is not an appropriate facing orientation, such as for following an animation path. A model that has had it’s “pointAt” orientation rotated to a new value can also have that rotation reset to it’s original value. To accomplish this, the global *model_orientation_list* is used to catalog all models whose orientation has been rotated. The *model_orientation_list* is a property list, with the name of the model as the “property”, and a property list containing the vectors of the original “facing” direction and the “up” direction. If a “pointAt” orientation is reset, the original values are retrieved from the list, applied to the model, then the model’s information is removed from the *model_orientation_list*. The “pointAt” orientation handling scripts are:

- **rotateModelOrientation()** – Rotates the facing and up direction vectors of the specified model about a specified angle by a specified axis. If the model is not already located in the *model_orientation_list*, this command will add this model with it’s existing facing and up values to the list.
- **RotateModelFromDirection()** – Rotates a model relative to itself about a specified axis by a specified angle, but will apply an opposite rotation to the “pointAt” orientation using the **rotateModelOrientation()** command, effectively keeping the “pointAt” orientation unchanged relative to the world coordinates.
- **ResetModelOrientation()** – Sets the specified model’s “pointAt” orientation to the original values possessed by the model when it was added to the *model_orientation_list.* That model reference is then removed from the *model_orientation_list.*

8.1.2 VISIBILITY

It is possible to remove a model from the 3D world, or to toggle the visibility of that model. This can be useful when a model is an integral part of the world, but does is not needed for a scene. When the visibility of a model has been set to “none”, Director will not expend any computational resources calculating the faces of that model. Models can be removed from a 3D scene in XDRIVE for the duration of the scene using the **disincludeModel()**, which sets the visibility of the model to “none”, and then removes it from the *current_model_list*, removing it from the list of recognized models.

To allow models be toggled between off and on states, the **turnModelOff()** and **turnModelOn()** commands are used. When turning a model off, the model is added to the global *invisible_model_list*, a property list storing model names that are currently turned off, and the visibility settings of those models prior to being turned off. The visibility of that model is then set to “none”. Turning a model on reverses the process by setting the visibility of a model to its original value – retrieving the property from the *invisible_model_list* – and then removing the model reference from the *invisible_model_list*. Valid visibility settings for models within Director are “none”, "front", "back", and "both". Front, back, and both refer to the side of face that will be rendered. Front-faced objects render a face when the face normal is pointing towards the camera, back-faced objects render a face when the face normal is pointing away. Both-faced objects render a polygon regardless of whether the face normal is pointing towards or away from the camera.

8.1.3 SHADERS

As a model's geometry defines the shape of the model that is rendered, the surfaces of the model are defined in Director by "shaders". More commonly known as "textures" or "materials", the shader controls the coloring, opacity, shininess, and reflection of a surface. A model's shader can be a solid color, or a mapped image, and the XDRIVE system is scripted to alter the shader of a model and animate those changes. These alterations include changing the color of an object, changing the opacity of an object – such as fading an object in and out, and highlighting an object (where a predefined color is superimposed upon a model in a cycled animation).

To preserve the shader information for a model, changes to the shader are not actually applied to the model itself, but instead are applied to an exact clone of that model. This model clone is created using Director's "clonedeep", which creates a duplicate of the model and duplicates of all attributes associated with the model, including the geometric resource and any shaders, ensuring that changes to a model will not inadvertently affect any other model that may share the same shader. It turns off the original model, and catalogs that model in the global *custom_shader_list* – a property list containing the names of all models that have had clones created for custom shaders, and properties pertaining to those shaders.

To create the clone of a model to receive custom shader events, the **createCustomShader()** command is used. The only input is the name of the model for which a custom shader is to be created. Once the model has passed the **verifyModel()** test, the model's clone is created using Director's "clonedeep" command. The name for the cloned model is the name of the original model, plus "_shadeclone". (This "_shadeclone" suffix is utilized for model recognition during click events, since models with no visibility will not register when the models under a point are logged.) The visibility of the original model is set to "none" effectively turning that model off. The cloned model is then defined as a child of the original model, ensuring that events that effect the original model will equally effect the cloned model. Finally, all of the shaders for the cloned model are logged, and the model is registered with the *custom_shader_list*, organized by the model name, and containing the cloned model name, the list of cloned shaders, and the previous visibility of the original model.

To reverse this process, the **removeCustomShader()** command is available. This command retrieves the original model information from the *custom_shader_list*, and restores it to the state it held before the custom shader was created. The cloned model and all custom textures (held by the shader list) are then removed from the scene and destroyed. Finally the model is removed from the *custom_shader_list*.

The following functions found in the "Shader Scripts" employ custom shaders to alter the visual properties of models:

- **setModelOpacity():** This function sets the level of transparency of a specified model to a specified percentage value. The model is verified, and checked against the *custom_shader_list*. If the model has not been registered with the list, the **createCustomShader()** function is evoked. The model is then added to the global *model_opacity_list*, a list of models that have undergone opacity events, and their current opacity value. The system then runs through all of the cloned model's custom shaders (referenced by the shader list retrieved from *custom_shader_list*) and sets each shader's opacity value to the specified percentage of it's original value. (In Director shaders, the opacity is referenced by the "blend" attribute).
- **changeModelOpacity():** This function determines the opacity percentage by performing a linear interpolation between start and end opacity values for a model. The opacity percentage is then altered in the model through use of the **setModelOpacity()** command.
- **FadeOffModel():** This function fades a specified model over a specified duration to an opacity value of 0, then turns the model off. Once it has verified the model's existence, it adds three events to the queue:
    - a change opacity event that will fade the model over the duration
    - a turn model off event that holds for completion of the fade event
    - a final opacity event that resets the model to its original opacity value, so that it is restored to its original state if it is toggled on again.
- **FadeOnModel():** This function works as the reverse of **fadeOffModel()**. Three events are added to the queue:
    - an event to reset the model's opacity to 0
    - a turn on model event
    - a fade-in event that changes the opacity to the original value over the duration.
- **ChangeModelColor():** This function operates similar to a combination of the **setModelOpacity()** and **changeModelOpacity()** commands, but rather than interpolating between opacity values by a percentage, color value of the shaders is interpolated. The specified model is verified, and if it does not exist within the *custom_shader_list*, the **createCustomShader()** function is invoked. If not already registered, it is cataloged with the global *model_color_list*, a property list holding the names of models that have undergone color change events, and the original color values of those models. The current color value between the start and end color values is calculated using the **interpolateColor()** command. Finally, the ambient and diffuse attributes for the shader are set to the interpolated color.

8.1.4 HIGHLIGHTING

The highlighting system is used to call attention to certain models by having the surface of the model pulse a specified color on and off, cycling over a set duration. Rather than add repeating pulse events to the event queue, models (and groups) to be highlighted are added to the global *model_highlight_list*. During playback, the **cycleModelHighlight()** command is invoked. This command determines the percentage completion of the cycle using the modulus operator of the defined cycle time against the current time, ensuring that all highlighted objects pulse with an identical rhythm. When they are added to the list, models are checked against the *custom_shader_list*, and a custom shader is created for the object if one does not already exist.

8.2 GROUPS

Groups, as previously state, are simply containers of objects – models, lights, cameras, and even other groups. Groups contained within the model are cataloged using the global *current_group_list.* This list is used by the **verifyGroup()** command to determine the validity of a named group.

Groups can either exist prior to playback in the imported file, or they can be created by the system. The XML <group> tags will create a group that includes a list of named objects. The <ungroup> tag will subsequently delete that grouping, but all objects will remain intact.

To create a group, the **createNewGroup()** simply verifies that the name is not already taken, then creates a group of that name and registers it with the *current_group_list*. The command **addModelToGroup()** includes a model within the group by setting the model as a child object of the group itself. The **addLightToGroup()** command operates in the same manner, but adds a light object to the group instead.

To remove items from a group, the **deleteModelFromGroup()** and **deleteLightFromGroup()** commands take a named model or light and reset the parent of that object to the group "World", which is the top level of the 3D scene. To ungroup an object, the **destroyGroup()** command first resets the parent of all child objects to the group "World", then issues a "deleteGroup" command, and removes the group from the *current_model_list.*

Groups essentially define a hierarchy, acting as a parent object to the child objects contained within. Motions and rotations events received by groups affect all objects contained within the group equally – they all move or rotate as one object relative to the pivot point of the group. The position, rotation, and scale commands for groups function similar to those for models. They are:

- **getGroupPosition()**

- **getGroupRotation()**
- **getGroupScale()**
- **setGroupPosition()**
- **setGroupSelfRotation()**
- **setGroupWorldRotation()**

In addition to having a position and direction, groups also have a "pointAt"orientation. The scripts to handle group pointing function similarly in method to those for models, except these use the global *group_orientation_list* instead of *model_orientation_list*. These functions are:

- **getGroupDirection()**
- **pointGroupAtPosition()**
- **pointGroupAtDirection()**
- **rotateGroupFromDirection()**
- **rotateGroupOrientation()**
- **resetGroupOrientation()**

Opacity, color, and highlight events only affect models, so when these events are applied to groups, they are executed by simply re-issuing the command for each model contained within the group.

- **changeGroupOpacity()**
- **changeGroupColor()**
- **highlightGroup()**
- **unhighlightGroup()**

8.3 LIGHTS

Lights have no renderable geometry of their own, rather they become visible through their interactions with models within the 3D world. Lights are cataloged using the global *current_light_list*, and can be validated against that list using **verifyLight()**. Like models and groups, they have positions and rotations, as well as "pointAt" orientations. The commands for handling position, rotation, scale, and direction for lights are:

- **getLightPosition()**
- **getLightRotation()**
- **getLightScale()**
- **setLightPosition()**
- **setLightSelfRotation()**
- **setLightWorldRotation()**
- **getLightDirection()**
- **pointLightAtPosition()**
- **pointLightAtDirection()**

- **rotateLightFromDirection()**
- **rotateLightOrientation()**
- **resetLightOrientation()**

Lights do not have opacity values or textures, only color values. Color changes for lights are handled using the same method for changing colors of model textures – linear interpolation of color values between the start and end values using **interpolateColor()**. For a light to "fade", the intensity of all three color channels are altered over a duration to approach a value of 0. When all three values are zero, the resulting light color is black, or no added intensity to models in any of the three model channels. However, that light still exists within the system, and is still calculated every time the frame is rendered – a calculation that can seriously hinder playback performance when more and more lights are included in the scene. To prevent these unnecessary calculations, when a light is turned "off" by the system, the light's properties are recorded in the global *off_light_list*, and the light is then deleted from the model. To then turn a light "on", a new light is created, and the light attributes contained in the *off_light_list* are assigned to it, essentially creating a new copy of the original light. The functions to handle visibility, color changes, and fade events are:asf

- **disincludeLight():** Deletes a specified light from the 3D scene, and removes it's reference found in *current_light_list.*
- **turnLightOff():** Similar in function to **turnModelOff()**, this function will fully delete the light, after recording the properties and transformation of that light in the *off_light_list.*
- **turnLightOn():** Counters the **turnLightOff()** script by creating a new light, and applying the properties of the now-deleted original light that were stored in the *off_light_list.*
- **changeLightColor():** sets the color of a light or the background to a percentage value between a start and end value. This function can also access lights that have been "turned off" or deleted from the model, by searching for an instance of the light in the *off_light_list* and setting the color of that lights properties.
- **fadeOutLight():** Similar in function to **fadeOffModel()**, this function send three events to the queue:
  - change the color to black – rgb(0,0,0) – over the specified duration
  - turn the light off
  - change the color to the original value.
- **fadeInLight():** Counters the **fadeOffLight()** command. Uses three events sent to the queue:
  - change the light color (in the *off_light_list*) to black – rgb(0,0,0)
  - turn the light on
  - change the color to the final value over the specified duration.

8.4 CAMERAS

The XDRIVE camera is the only camera that can be referenced by the system during playback. The camera system (herein referred to as the "camera") can be positioned and pointed, but rotation scripts are more complex, since there are certain restrictions that apply to the motion of the parent model (herein referred to as the "camera object model') and the actual camera-type object used as the viewpoint for the 3D sprite (herein referred to as the "actual camera"). A reference to the camera object model is stored in the system by the global *cameraobject* and this reference is used as a means of retrieving properties and applying new values. The scripts for position and direction of the camera are:

- **getCameraPosition():** Returns the position of the camera object model
- **getCameraTilt():** returns the tilt angle of the camera, relative to the XY plane – a value ranging between negative 90 and 90 degrees. This is accomplished by reporting the angle of the actual camera in relation to the parent camera object model
- **getPanDirection():** returns a direction vector of the camera. This is accomplished by moving the camera object forward 1 unit relative to itself, recording that position, then moving back to the original position. The old position is subtracted from the new position to create the direction vector
- **getPanAngle():** returns the angle of the camera pan, relative to the positive Y axis rotated about the Z axis – a value ranging between negative 180 and 180 degrees. The absolute value of the angle is found by determining the camera's pan direction then calculating the angle between that direction and a direction on the positive Y axis. The pan direction is then evaluated for the angle between it and both the positive and negative X axis to determine if it is a positive or negative rotation about the Z axis.
- **getCameraDirection():** returns the current directional vector of the actual camera. This is calculated much in the same manner as the **getPanDirection()** command, only the camera object model is now rotated to match the camera's tilt angle before it is moved 1 unit and the new position recorded. The camera object model is then moved back to it's original position and rotation, and the directional vector is calculated by subtracting the original position from the new position.
- **setCameraPosition():** sets the position of the camera to the specified coordinates. This is accomplished by setting the camera object model to the new position by setting the "transform.position" attribute of the *cameraobject* reference. The camera object model acts as a parent to the actual camera, so the same position is applied.
- **translateCameraWorld():** moves the camera relative to the current position by the specified translation vector – an XYZ relative coordinate vector. The camera object model is repositioned using Director's "translate" function.
- **tiltCameraAbsolute():** tilts the camera to an absolute angle between negative 90 and 90 degrees. This is accomplished by setting the angle of rotation of the actual camera relative to the parent camera object model. Angles falling outside of the

allowed range will be discarded, and the minimum or maximum value will be applied instead. (Currently, this function is not invoked by any other function, but is still included)

- **tiltCameraRelative():** tilts the camera by the specified angle, relative to the current tilt angle. This is accomplished by retrieving the current tilt angle, then adding the degree increment for the final target. If this new angle falls outside of the allowable range (negative 90 to 90 degrees), the new value will be overwritten with the minimum or maximum value accordingly. Finally the actual camera's rotation relative to the parent camera object model will be set to the new value.
- **panCameraRelative():** rotates the camera about the positive Z axis by the specified angle increment, relative to the current degree. The camera object model is rotated by applying the "rotate" command to the *cameraobject* reference. The camera object model acts as a parent to the actual camera, so the same rotation is applied.

8.4.1 CAMERA POINTING

The camera assembly is comprised of two objects that share the duties of rotation, each exclusively responsible for it's own rotation. The camera object model rotates about the Z-axis, turning to face a point along the XY plane. The actual camera only rotates up and down (relative to the camera object model), which provides the means for facing a point located vertically above or below the XY plane. This up and down rotation is restricted to prevent rotations that would flip the camera's viewpoint to one where the 3D sprite's "up" direction would be facing opposite the 3D world's "up" direction, effectively displaying the world as if the camera were upside-down. The camera is also restricted from tilting from side to side. Using this two-rotation method, the camera can face any position in the 3D world not currently occupied by the camera itself.

Due to these rotation restrictions, traditional rotations cannot be directly applied to the camera. Instead, rotations to the camera are accomplished by determining the camera's direction vector – a point represented by a position relative to the camera along the current facing direction – and rotating that direction vector around point (0,0,0) by the specified axis and angle, which results in a new direction vector. This new direction vector is added to the current camera position resulting in a final location which the camera is then directed to point at, using the 2-step rotation. The camera rotation functions are:

- **getPanAndTiltDifference():** This function determines the necessary relative pan and relative tilt angles to point at a specified location.
    - The relative pan angle to face the XY coordinates of the new point is calculated using the following steps:
        - The camera position and pan direction are retrieved
        - The new XY direction vector is calculated by subtracting the current camera X and Y coordinates from the X and Y coordinates of the specified position.

        - The angle between the current pan direction and the new pan direction is calculated
        - The angle is determined to be clockwise (positive pan angle) or counterclockwise (negative pan angle) by comparing the angle between the new direction and the camera's direction when rotated 90 and –90 degrees.
    - The relative camera tilt angle needed to face the Z coordinate of the new point is then calculated using the following steps:
        - The tilt direction vector to the final point is calculated by subtracting the current camera position from the new position.
        - The absolute tilt angle is determined by calculating the angle between the tilt direction and the new XY direction vector. If the tilt direction vector's Z coordinate is less than zero – meaning that the new position has a lower Z value than the camera position – the angle is considered negative.
        - The current camera tilt angle is evaluated, and the relative tilt angle is calculated by subtracting the current tilt angle from the new tilt angle.
    - The pan and tilt angles are returned from this function as a property list.
- **pointCameraAtPosition():** This function will point a camera at the designated position. This is accomplished by retrieving the relative pan and tilt angles from the current camera direction using **getPanAndTiltDifference()**. The returned pan and tilt angles are then applied to the camera using **panCameraRelative()** and **tiltCameraRelative()**.
- **pointCameraAtDirection():** This function points the camera at a specified direction relative to the current camera position. This is accomplished by adding the direction vector to the current camera position, and using the **pointCameraAtPosition()** command to point the camera at the resulting position.
- **rotateCameraByAngle():** This function will rotate the camera's direction a specified angle about a specified axis. This is accomplished by determining the direction of the camera (returned as a relative direction vector) then rotating that vector point around a point (0,0,0) resulting in the new facing direction. The camera is then pointed at that direction using the **pointCameraAtDirection()** command.
- **RotateCameraFromDirection():** This function will rotate the direction used as the "facing direction" about the Z axis by a specified angle. This is used to change the camera's orientation – the camera system's equivalent of the "pointAt" direction – for situations where a camera will be following a path but should be facing an angle relative to that path. An example of this would be turning the camera to the left and right while following a walkthrough path. Instead of using Director's "pointAt" orientation, the camera's facing direction is stored as an absolute pan angle by the global *camera_orientation*. The orientation change is accomplished by performing a relative pan on the camera, then replacing the *camera_orientation* value with the pan angle subtracted from the current value.

- **ResetCameraOrientation():** Sets the *camera_orientation* value to 0, which sets the camera's path facing direction to equal the view facing direction.

8.4.2 MULTIPLE CAMERAS

Director will support multiple cameras within a 3D world, and supports multiple cameras simultaneously displaying the same 3D world, as was used in the KAJO presentation. This process can be costly on system performance, just as larger 3D sprites can affect performance, due to the additional rendering that must occur. Since XDRIVE implements a custom camera with specific restrictions, no scripts are included at this time to access, edit or alter any cameras contained within a scene other than the main camera, or to include more than one 3D sprite in the presentation layout.

8.5 OBJECT SCRIPTS

To simplify XML tags as well as prevent errors resulting from incorrect or missing object type information, the functions in "Object Scripts" are included to determine the type of a named object and issue commands to a named object without knowing the type.

VerifyObject() is used to determine the existence and type of a named object. It operates by running verifications of the object name against all types, returning the first type it positively encounters. The checking order is: models, groups, lights, camera. If a named object is found to be within one of those catalogs, that type will be returned. If no named object if found within the catalogs, then no type is returned. For this reason, it is important not to have objects of different types with identical names, otherwise the incorrect type may be returned and the action inadvertently performed on the wrong object.

The remaining object functions simply verify the object type using **verifyObject()**, then issue the object specific command to that named object. These functions are:

- **getObjectPosition()**
- **getObjectDirection()**
- **pointObjectAtPosition()**
- **pointObjectAtDirection()**
- **pointObjectAtPathDirection()**
- **resetObjectOrientation()**
- **highlightObject()**
- **unhighlightObject()**

9.0 ANIMATIONS

Objects within the 3D world have three properties of transformation – position, rotation, and scale. The animation of these properties of an object is achieved through the incremental alteration of these transformations through the appropriate interpolation of the event. XDRIVE does not currently feature the ability to animate scale, only position (herein referred to as a "motion event") and rotation (herein referred to as a "rotation event"). Motion and rotation events feature the ability to utilize ease-in / ease-out curves, the acceleration and deceleration of the rate of event completion, as was outlined previously in section on Interpolation. These accelerations and decelerations are accomplished in events either through acceleration and deceleration settings, or through the use of speedcharts that are stored in motion and rotation events in the event queue. It is important to keep in mind that these interpolations are only interpolations of the percentage of event completion, which is not always a direct relation to the percentage of event distance, as is the case with "spline paths", which are covered later in this section.

9.1 ROTATION EVENTS

The rotation events recognized by XDRIVE are broken into two categories – object rotations affecting the rotation or orientation property of models, groups, or lights; and camera rotations, affecting the facing direction of the camera system. All rotation events are relative events, that is, they do not define a discrete end rotation property of an object's transformation, but rather a rotation upon an object's transformation at the time of execution. Rotation events are added to the event queue by the **setupRotation()** command, which parses each command for errors, builds the event with the proper command, and adds it to the queue.

9.1.1 OBJECT ROTATION

There are three types of object rotation events.

- object self rotation
- object world rotation
- object orientation rotation

These events only alter the rotation property of an object's transformation or "pointAt" orientation, not the object's position. In these events, objects or their facing orientations are incrementally rotated a set number of degrees around a specified axis passing through the pivot point of that object. Objects undergoing a self rotations are rotated about an axis defined with respect to the object's own coordinate system, while world rotations are rotated about an axis defined with respect to the world coordinate system. One example of these rotations would be the wheels of a car as the car travels in a circle. The car is rotating around an axis defined in world space, but spin axis of the wheels is constantly changing with respect to world space. In this case, the axis of rotation the wheels is defined by the wheel itself – this is self rotation.

Object self, world, and orientation rotations are interpolated using the **interpolateObjectRotation()** command. This command determines the percentage of rotation that should be complete by testing the current time against the acceleration / deceleration curves using the **findDistanceByTime()** command. It then determines the increment angle to rotate the object by subtracting the previous iterations completed percentage by the current percentage. Finally, depending on the rotation and object type, it issues the proper command to rotate the object or object orientation – for models: **setModelWorldRotation(), setModelSelfRotation(), rotateModelOrientation()**.

9.1.2 CAMERA ROTATION

There are also three types of camera rotation events:

- camera angle
- camera direction
- camera orientation

Camera rotations differ from object orientations since the camera system restricts the rotations that can be performed. Camera angle rotations perform relative rotations on the pan and tilt rotations. Camera direction rotations perform rotations by rotating the direction of the camera around the defined axis. Camera orientation rotations will pan the "pointAt" orientation of the camera. These rotations are interpolated using the **interpolateCameraRotation()** command, which is called directly from the event queue. This command determines the percentage of completion by testing the elapsed event time against the acceleration and deceleration curves using **findDistanceByTime()**. It then determines the appropriate angle (or angles) to rotate the camera or camera orientation, and executes the incremental rotation using the camera rotation commands: **panCameraRotation(), tiltCameraRotation(), pointCameraAtDirection(),** and **rotateCameraFromDirection()**.

9.2 MOTION EVENTS

Motion events are animations that achieved by altering the position attribute of the transformation of an object over time. There are three types of motion events recognized by the XDRIVE system:

- linear motion
- orbit motion
- spline motion

Motion animations often consist of more than one motion event, which is especially the case with spline motions, which are broken up into individual events for each segment connecting two control points along the motion path. To properly interpolate these animations by applying a single timeline across multiple events, it becomes necessary for the system to pre-determine the overall distance that will be traveled across the entire event, and apply start times and durations to each event within the system accordingly. For the system to group these individual animation events as one command, motions are defined in the XML document and passed to the **setupMotion()** command as sequences of motions. Since the start position of an object in a motion sequence will not always be known in advance, the start position attribute for motion sequences can be defined as **#here** – a placeholder value that informs the system that the current position of the object at the start of the motion event is the position to be used as the start point for the sequence of motion events. However, since the **setupMotion()** command is used to evaluate the overall distance traveled over the motion path, and since the start value may not be defined until the moment the sequence begins, motions are not evaluated at the start of a scene playback. Instead, when the XML system parses a motion event, it catalogs it using the global *motion_catalog*, a property list holding the unique name event, and the XML of the motion sequence. A placeholder "call motion" event is added to the event queue, set to execute at the motion sequence start time. Upon execution of the call, the motion sequence XML is retrieved from the catalog, parsed, and is then passed to **setupMotion()** which evaluates the sequence – adding the distances traveled over each segment of the sequence, calculating start times and durations for each segment, and calculating the speedcharts for each event as they relate to the overall acceleration / deceleration curves – and then creates the individual motion events for each step of the sequence which are stored as attributes for the overall motion in the event queue.

9.2.1 LINEAR MOTION EVENTS

In linear motion events, objects travel in a straight line from a start point to an end point. These events are the simplest motion to calculate – the position of the object within space is a direct linear interpolation between the start and end coordinates. The event is interpolated to the percentage defined by the elapsed time over duration, or by using the speedchart created by the acceleration / deceleration curves. They are also the easiest to evaluate for distance traveled, using Director's "distanceTo" command which evaluates linear distance between two given points. There are two variations of linear motion events – absolute and relative. Absolute linear motions have an end point defined in world space coordinates and interpolated as a percentage distance between the start and end point. Relative linear motions define a translation to be executed regardless of other motions that may execute, interpolated as increments of the total motion. An example of relative motions at work would be an object starting at (0,0) and moving to a point 10 units in the X direction. If a relative motion event of (0,10) is applied to the object at the same time, moving it 10 units in the Y direction, the final location of the object will have a value of X=10, Y=10. If an absolute motion event is were to be applied to the same object with a value of (0,10), the final position of the object would be (0,10), since the absolute motion event defines a specific world coordinate as an end point.

When a linear motion event is interpolated, the event percentage completion is calculated, and a world coordinate is calculated and applied. For absolute linear motion events, the **getLinearPoint()** command is used to calculate the correct point. For relative linear motion events, a direction vector increment is determined by applying a percentage of instance completion (arrived at by subtracting previous completion percentage from current completion percentage) and using Director's "translate" command to apply that direction vector to the object.

- **current position = current % * (end position – start position) + start position**

(Figure 9.1)

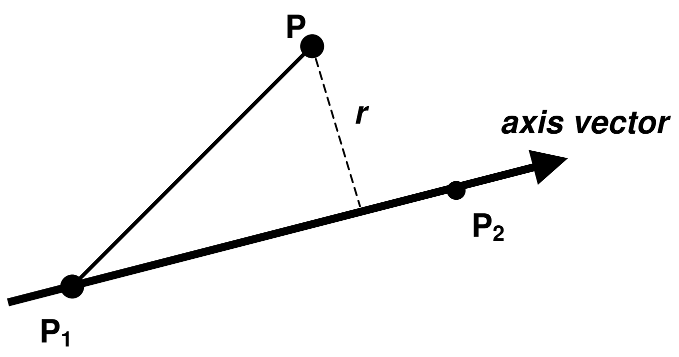
9.2.2 ORBIT MOTION EVENTS

Orbit events rotate an object by a specified angle about a specified axis that passes through a specified pivot point. Orbit events do not have a specified end point, rather the end point is determined by the rotation attributes. The end position (and all positions interpolated along the arc path) are calculated by creating an empty transformation, setting the position attribute of the transformation equal to the world position coordinate of the object, then using Director's "rotate" command to perform the defined rotation and returning the final position of the temporary transformation. This will only alter the position of the object – it will retain the same orientation. Orbit motions feature a "rotate" attribute that when set to TRUE will alter the orientation of the object transformation by rotating the object an equal number of degrees about the specified axis using the **setObjectWorldRotation()** command.

Determining the distance traveled for orbit rotations requires additional calculation. The method for determining the distance of an arc path is:

- **(2π * arc radius) * (arc angle / 360)** (Figure 9.2)

However, the arc radius for orbit rotations is only equal to the distance from the orbit pivot point to the object position when the orbit pivot point falls on the plane defined by the resultant arc. When the object position and orbit pivot are collinear along the pivot axis, the distance traveled will be 0. When the orbit pivot and the rotation arc are not coplanar to the distance from the object position to the axis of rotation, or collinear with the axis of rotation, the radius of the arc will be less than the object to pivot point distance, and greater than zero. To determine this radius, the distance of the object perpendicular to the rotation axis is found by calculating the cross product of the vector from the pivot point to the object position, and the vector of the axis of rotation , then dividing the length or magnitude of the resulting vector by the magnitude of the vector of the axis of rotation. This is illustrated in Figure 9.3.



- $r = | (P - P_1) \times (P_2 - P_1) | \; / \; | (P_2 - P_1) |$

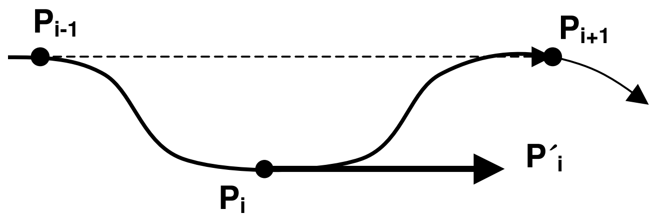
  Where:
  - P = Object Position
  - P_1 = Pivot Position
  - $P_2 = P_1$ + Axis Vector
  - r = radius

[Parent, p. 425]

(Figure 9.3 – Determining the Radius for a Non Coplanar Rotation Arc)

9.2.3 SPLINE MOTION EVENTS

To accommodate smooth motion animations between a series of points along a path, XDRIVE utilizes a custom spline system to calculate interpolations of a curved path of motion through defined control points. These paths use cubic polynomial equations – in this case the Catmull-Rom equation – to calculate points along the path. There is a value to using these complex calculations between points rather than using simple linear interpolation. Whereas linear interpolation would result in straight segments from point to point, the spline path is a smooth curve that passes through each point at a direction related to the previous check point and next check point. This direction can be calculated and applied as the target direction for an object or camera moving along that path.



(Figure 9.4 – Diagram of a Spline Path Segment)

Spline paths are defined in the XML file as a motion segment, but to properly interpret them within the system they are divided further into one motion event for each position after the start point of the spline path. Each path segment contains positions for the start and end control points, as well as the start point of the prior segment, and the end point of the following segment.

The positions and tangent vectors of points along the curve are calculated through functions applied to the relations of those four points. Variations in those relations can result in fluctuations of the magnitude of tangent vectors, meaning that the amount of distance covered will vary over equal increments of completion at different locations along the curve; essentially an object moving at a steady rate of segment completion will appear to speed up and slow down depending on the relation of the control points.

To maintain a constant motion speed along a spline path, the percentage of an event completion must be adjusted against a percentage of path completion so that the proper rate of path completion is used, resulting in an equal amount of distance passed over an equal time increment. For each segment of the curve, a “distance chart” is calculated by splitting the path into smaller segments to evaluate the distance traveled at defined increments of completion.

The distance chart is a list of distances traveled at 20 equal increments of completion. Each 5% increment is evaluated by calculating the position at that cumulative percentage of completion, then calculating the distance between that position and the position at the previous increment. This distance is added to the distance recording of the previous increment to obtain the current distance traveled at this percentage of completion. It is important to note that these distances are NOT normalized – they are world coordinate distances. Once the distance chart has been calculated, the final distance of the path segment is defined by taking the last item in the distance chart – the distance traveled at 100% completion.

The four control points and distance chart used for each path segment are stored as attributes for the motion segment, which is interpolated during playback using the **interpolateSplineMotion()** command. To smooth the motion along the path, the percentage of event completion (time elapsed) is converted to the percentage of path completion using the **getDistanceChartParametric()** command. This function takes a value corresponding to the distance that should be traveled (determined by total distance times percentage completion) and compares it to each value in the distance chart. Once it finds a distance value greater than the current traveled distance, it determines at what percentage between that value and the previous value the current distance value falls. That percentage is then returned and used as the percentage of path completion for the spline calculation.

9.3 MATRIX FUNCTIONS

The cubic polynomial equations to determine points along a spline path employ matrix algebra for speedy calculation. To handle these calculations, a series of custom matrix functions were developed to add this ability to the program. These functions allow the system to create, and verify these two-dimensional numeric arrays:

- **newMatrix():** creates a two dimensional numeric array with a size of *m* rows by *n* columns, with each entry set to a float value of 0.0.
- **validateMatrix():** checks a specified matrix to verify that it is properly structured as a two dimensional array, and that each value is numeric. Integer values are converted to float values.
- **getMatrixParams()**: returns the number of rows and column in a specified matrix, used for validating matrices for multiplication.

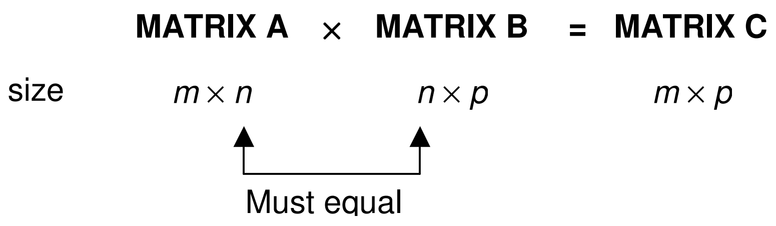
There are two types of multiplications for matrices used – multiplication by a scalar value, and multiplication by another matrix. Scalar multiplication of a matrix will multiply each number within the matrix by the same numeric value. The **multiplyByScalar()** command will receive a matrix object and a scalar value, and return a new matrix after it has multiplied each value by the scalar.

Multiplying a matrix by another matrix is a more complicated calculation. To multiply matrices together, each entry within the *i*th row of the first matrix is multiplied by the corresponding entry in the *j*th column of the second matrix. These multiplied elements are then added together to form the row *i* column *j* entry of the resultant matrix. [Parent, p. 410] This equation is expressed as:

- $\mathbf{C_{ij} = (A_{i1} * B_{j1}) + (A_{i2} * B_{j2}) + (A_{i3} * B_{j3}) + \ldots (A_{in} * B_{jn})}$

(Figure 9.5)

To multiply matrices, the number of columns in the first matrix (A) must be equal to the number of rows in the second matrix (B).



(Figure 9.6 – Matrix Multiplication Diagram)

The **multiplyMatrices()** command takes two matrices and returns the result of their multiplication. The two matrices are first validated, then the "inside dimensions" are verified to be equal. If these conditions are met, a new matrix is created with the resultant dimensions, and each value is populated by calculating the equation in Figure (INSERT FIGURE NAME HERE) for each value of *i* and *j*.

9.4 SPLINE INTERPOLATION

A position along a spline (linear interpolation along a parametric curve) can be evaluated through matrix multiplication as:

- $\mathbf{P}(u) = \mathbf{U}^T\mathbf{MB}$ (Figure 9.7)

  Where:
  - $\mathbf{P}(u)$ is the interpolated value of u percentage along the curve
  - $\mathbf{U}^T$ is a one dimensional matrix variable: $| u^3 \; u^2 \; u \; 1 |$
  - **M** is the coefficient matrix, defined by the spline type
  - **B** is the geometric information matrix consisting of values from the four control points for the spline path

[Parent p. 456]

P(u) can be thought of as a function that is applied to each dimension. To determine a position in 3D world space, this function must be executed three times, one for each dimension. The U^T and M values will remain constant, but B will have a different value for each dimension – one for the X dimensions, one for the Y dimensions, and one for the Z dimensions – as shown is in Figure INSERT FIGURE NUMBER HERE.

- $\mathbf{point(\,x, y, z\,) = (U^TMB_x, U^TMB_y, U^TMB_z\,)}$ (Figure 9.8)

Across a path segment – between two control points – M and B will remain constant. Before the path segment is evaluated for the distance chart system, the values for MB_x, MB_y, and MB_z are calculated and stored as attributes for the motion segment. During playback, these MB values are retrieved as attributes, and used to execute this equation to find a point along the curve path.

9.4.1 CATMULL-ROM SPLINE

The Catmull-Rom spline curve is used primarily for simplicity – it is quick to calculate, and the only points required to manage the shape of the curve are the path control points. While other Bezier curve splines use additional positions to define the tangent vectors of the line at control points, the Catmull-Rom calculates the tangent vector (P'_i)of an inside control point (P_i) as ½ of the direction vector connecting the previous control point (P_{i-1}) to the next control point (P_{i+1}). [Parent, pp. 458-459]

- $\mathbf{P'_i = \frac{1}{2} (P_{i+1} - P_{i-1})}$ (Figure 9.9)

In addition to the start point and end point, the equation needs the control point prior to the start point to calculate the start point's tangent vector, and the control point after the end point (P_{i+2}) to calculate the end point's tangent vector. For the Catmull-Rom equation:

$$\mathbf{U^T} = \left[\, u^3 \; u^2 \; u \; 1 \, \right]$$

$$\mathbf{M} = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \qquad \text{(Figure 9.10)}$$

$$\mathbf{B} = \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

[Parent, p 459]

These equations are employed by the system using the "Catmull-Rom" scripts, which include the following functions:

- **CR_MSetup():** Creates the M value matrix
- **CR_GetSegmentMB():** Creates the three B values by defining matrices with the x, y, and z values for the specified prior point, start point, end point, and next point, then multiplies each with the preset M to determine the MB_x, MB_y, and

MB_z values which are then stored in the motion segment attributes for later calculation.

- **CR_GetPoint():** Returns a position value by creating the U^T matrix from the specified percentage value, and multiplying it by the specified MB value
- **CR_GetDerivative():** Returns a derivative which defines the tangent vector at a point. The equation for this is:
    - $P'(u) = U'^T MB$
      Where: $U'^T = | \ (3 * u^2) \ (2 * u) \ 1 \ 0 \ |$
- **CR_GetSegmentDistanceData():** Calculates and returns the distance chart used for event interpolation.

10.0 TARGETING

The targeting system allows XDRIVE to rotate an object to "point" at a specified direction or position, regardless of the position of the object. As long as a target is applied to an object, it will continue to orient itself towards that direction or position, even when changes occur in the position of the object or the target. This allows relations of objects to be defined and to persist without having to know discrete positions or facing directions. The most obvious of these targets is the path target, where an object will follow a motion path currently applied to it, such as a camera following a spline path as it makes its way through the world.

Objects turn to face their target by rotating the object so that the "pointAt" orientation – the self-coordinate relative vector that defines the object's facing direction – aligns with the assigned world coordinate direction vector or the world vector that connects the current object position and the position of the object's target. There are four types of targets:

- **Point:** Objects face a defined world coordinate
- **Direction:** Objects face a world-relative direction vector
- **Object:** Objects face the current position of another named object
- **Path:** Objects face the direction vector defined by the current motion path

When a target is assigned to an object, that target is added to the global *object_target_list*, a property list that catalogs all current object targets. During each call to the command loop, the **updateObjectTargeting()** command is called. This function then runs through each object currently in the *object_target_list*, evaluate the target and issue a **pointObjectAtPosition()** or **pointObjectAtPoint()** command which is then passed to the command associated with the appropriate object type. This repeated function ensures that the object will continue to face the assigned target, even as changes occur to the object or target's transformations.

For object targets with a type of "point" or "direction", the target value remains constant, and is retrieved from the *object_target_list*. Object targets with the "object" type are designed to face the current position of another named object. Each time the object target is updated, the position of the target object is evaluated using the **getObjectPosition()** command. As long as the targeted object is valid, the main object will be turned to face towards a point that corresponds with that position.

The values of object targets that are of the "path" type are assigned as direction vectors, and will often vary over time, depending on current motions being applied to that object. To keep the object targets consistently updated when undergoing motion events, the appropriate facing direction is evaluated during each execution of the motion interpolation scripts, and then sent to the **setDirectionalTargetPath()**, which checks the *object_target_list* to see if that object is currently assigned a path target, and if so adds the new direction value to the attributes.

Object targets are added to the *object_target_list* through the use of the **setDirectionalTarget()** command, and can be removed from the list by the **deleteDirectionalTarget()** command which is issued when a target of type "none" is issued by the event queue, effectively telling the targeting system that the duration of that target has expired.

10.1 EVALUATING PATH DIRECTIONS

There are three types of motions – linear, orbit, and spline – each with their own method needed to evaluate the path direction of an object to be updated in the targeting system. Linear motions are the easiest to evaluate, as their path is defined by a straight line connecting the start and end points of the motion event:

- **direction = end position – start position** (Figure 10.1)

Orbit motions calculate the path by using the **getTangentVector()** command, which finds the current point along the arc path, then rotates that point an additional 90 degrees around the same axis to a point perpendicular to the radius vector of the arc at the current point. The pivot point is then subtracted from this new point to define a direction vector. If the current angle rotation is negative, the opposite vector is returned instead.

Spline motions are also simple to evaluate – the tangent vector of a spline path at point P(*u*) is the vector P′(*u*) – the derivative of P(*u*). The command **CR_GetDerivative()** returns the tangent vector by calculating the following equation:

- $\mathbf{P'(\mathit{u}) = U'^{T}MB}$
  Where: $\mathbf{U'^{T} = |\ (3 * \mathit{u}^2)\ \ (2 * \mathit{u})\ \ 1\ \ 0\ |}$ (Figure 10.2)

10.2 TARGET TRANSITIONS

Target events also have a duration so that when a target is applied to an object, the object's turn to face the target direction is animated, instead of simply snapping to the new facing direction. During a target transition, the object is added to the *object_target_list*, but with a target type of "transition" which allows the target to exist within the list, but not evaluate during the **updateObjectTargeting()** command. Instead, the **interpolateTargetTransition()** command evaluates the current direction of the object, the direction of the target, and the percentage of event completion, and interpolates the current facing direction using the **interpolateDirectionalRotation()** command. This command determines the angle between the two vectors, and the cross product of the vectors, which defines the axis of rotation. The direction is then rotated the percentage increment of the angle between the two vectors, and the resultant vector is applied to the target list.

11.0 TRIGGERS

The “trigger” system provide the interactive component of XDRIVE – allowing the user to interact with the content during playback. The “triggers” are commands or series of commands stored within the system that wait for a user-activated event to launch. Once the user-activated event occurs, a call is made to the trigger system to add the commands into the event queue.

There are two types of user-activated events (herein referred to as “trigger events”) that are recognized by the system – click trigger events, and proximity trigger events. These trigger events can be set to execute only once, or to execute every time the trigger event is activated. Trigger events also have a possible lifespan, expiring after a set duration has passed. Multiple trigger events can be assigned to the same trigger, calling the same stored commands whenever any of the trigger events are activated.

Click trigger events are called when the user “clicks” a model in the display that is registered with the click trigger catalog, and are evaluated any time the mouse button is pressed down while positioned over the 3D sprite. Model “clicks” are evaluated by determining the list of models underneath the mouse position (using Director’s “modelsUnderLoc” attribute) and issuing a call trigger command for the model closest to the camera, and is not currently turned off. If that model has a click trigger event assigned to it, it will execute. Click trigger events are cataloged using the global *click_trigger_list*

Proximity trigger events are called when the position of the camera falls within a specified distance of the position of an object to which the trigger event is assigned. The distance of the camera between all models with proximity events is updated and evaluated each frame, and tested against the assigned proximity distance. Proximity events are stored using the global *prox_trigger_list*. To keep proximity trigger events from being repeatedly activated as each frame renders, proximities have an “on” and “off” state. All proximity trigger events begin in the off state. Once the camera falls within the proximity distance, the event is triggered, and the proximity trigger event is now considered to be “on”. Now when the proximity check occurs, it will not call the trigger – instead it waits until the camera is once again outside of the proximity distance, at which point if the trigger event was a one-time-only event, it is removed from the proximity trigger list. If the trigger event is set to repeat, the proximity event is returned to the “off” state. There is an optional attribute for proximity events to call a second trigger when the camera moves outside of the proximity. This “leave trigger” can be used for a variety of purposes, such as resetting any changes that were made by the trigger called when the proximity event was first triggered – for example, if a proximity trigger event turns on a light, the leave-trigger event can turn the light back off. Proximity on and off states are managed by moving the model reference between the global lists *current_prox_on* and *current_prox_off*.

Triggers – the set of commands that are executed when a trigger event calls – have a unique name, which allows for multiple trigger events to call them. This name is assigned as the value of the “trigger” attribute of the trigger event.   The commands in named triggers are stored as the XML source they were imported as. When a named trigger is called, this source is sent to be parsed by the XML scripts and added into the event queue.

12.0 XML PARSING

The XDRIVE engine holds no predefinition of a presentation before it is loaded. No events are queued in the timeline, no 3D world file is stored, and no relation between objects such as triggers or targets are defined. Instead, the system relies solely on an external XML file to define the elements of a presentation. This single text document contains the settings, structure, and events necessary to build an entire presentation.

XML (eXtensible Markup Language) is a text based file format used for storing data. Data objects are stored as "tags" that include "attributes", and can nest additional data tags. Syntactically it is structured in a similar fashion HTML (HyperText Markup Language), the file format used for web pages that is read and interpreted by a web browser. XML was designed as an extension of HTML by providing a universal method for formatting data that could be read and understood by multiple systems. The power of XML is that there are very few predefined elements – it is an open format that can be defined using nomenclature specific to the data task at hand.

For the XDRIVE system, custom data tags and attributes were developed to be interpreted by the system. XML was chosen because of the ease of authoring, the common nature of the format, and because Director features XML parsing functions that allow quick access to data and data tag properties. The similarities in required functions between web pages and XDRIVE also lend some influence to the structure of the custom XML language. XML files can be written using any text editor. It is also possible that a presentation could be dynamically generated using a server-side scripting language such as PHP or ASP, or any program that can generate a text file from data.

For the system to recognize and load the XML file, it first determines the path to the file through a parameter defined in the EMBED and OBJECT tags used to include the Shockwave file in a web page. This file is located and imported as a text object by Director, which then converts the contents to XML data using the XML Xtra. Once loaded as an XML object, that object is parsed through the structure for settings and commands.

All XML tags contained within the <xdr></xdr> tags are considered to be within the XDRIVE system. At the top level, an XDR file is structured into two stages, the "setup" and the "body" of the presentation. The XML parser looks for these two tags, and will take their contents through each one's respective parser.

The purpose of the "setup" stage is to define the overall settings and layout for the presentation. The <preferences> tags are used to make all necessary adjustments to the navigation systems, such as keymapping, camera movement speeds, and the default navigation settings and restrictions. The <layout> tag defines areas on the screen that will reserve those locations for media elements. Each <region> tag defines a position and size within the screen, as well as a name for that area. When a media element in the presentation holds that name in it's region attribute, that element will appear in that position. The <worldlist> tags define names and file paths for multiple W3D files.

Scenes will use this name to refer to the Shockwave3D file that they will load. Although currently presentations are only capable of loading one file, future development may include the capability to load multiple files within the same presentation.

The “body” stage is where the presentation playback is defined. The <body> tag informs the system which 3D world will be used, which region the 3D world will be displayed in, and whether or not the diagnostic system is active. (The diagnostic provides updates during the loading of files, and during playback will display the camera’s current XYZ position, pan angle, and tilt angle – helpful for setting up predetermined camera points when authoring a presentation.

At the body stage, there are three tags that are recognized – worldsettings, preworld, and scene. The <worldsettings> tags define the overall settings for the 3D world file such as the position and rotation of the entirety of the world, the initial camera position, and the highlight and headlight settings. The <preworld> tags define object events that are to be executed before playback begins, such as camera fades, object transformations, shader effects, and the definition of custom groups.

The <scene> tags define the actual playback events of the presentation. Multiple scenes are available, and can be referenced by their custom name. If no initial scene is set, then the first scene encountered is parsed and executed. Scenes contain a <prescene> tag that is similar to the <preworld> tag – it contains object events that are to be executed prior to the playback of that scene. Other tags in the scene are all evaluated as object events, if their name corresponds with a recognized object event tag. A full list of these tags is available in Appendix A.

12.1 READING XML WITH DIRECTOR

Director imports XML as nested data objects with a parent / child structure. Each XML tag is considered an object, and properties of that tag can be tested and retrieved. The most important of these are the “name”, the “attributeName”, the “attributeValue” and the “child” properties. Names correspond to the name of the tag this data object embodies. The data object has a linear list of the names of attributes within the tag, and a corresponding linear list of values for those attributes, each accessible through the “attributeName” and “attributeValue” property. The “child” property is a linear list of the data objects that are nested within this particular tag.

To quickly evaluate a tag, variables for the appropriate attributes are predefined, either as default value, or as “void” where a test for required attributes will be performed. A repeat loop is used to test each attribute. If the attribute name tested value corresponds to an appropriate tag attribute (tested using a case comparison), the data in the attribute value is tested for format, and if found the attribute variable is assigned that value. After retrieving all necessary attribute values for a tag, the variables are tested and parsed, and the resulting commands are issued. For tags where child objects will be parsed, a similar repeat loop is used to test the name attribute for each child tag which is then tested in a case comparison and if a tag is a valid subset of the current tag, the data object of the child is sent to the appropriate parsing command.

While most commands are parsed prior to playback, certain XML commands, such as motions and triggers, are cataloged and stored as XML data until the system calls them, at which time will be parsed.

13.0 FUTURE DEVELOPMENT

The XDRIVE engine was developed with the idea that it would be able to easily support additions to the structure and capabilities. Non-specific event functions such as the timeline and event queue system allow for the extension of the system, and the XML scripts can easily be updated to handle new input. Director's offered components also allow for a wide range of possibilities for components and capabilities to be coded into the system.

The following items comprise a list of ideas for the future development of XDRIVE, although possibly future expansions are in no way limited to only the items on this list.

- **Images & Text Support**

  The importation and display of image and text objects, as well as the interaction with those events would greatly enhance the multimedia experience and provide greater interaction with the content. For the Katrina Jones Choi's thesis project – KAJO, a virtual grocery store – images were also included as part of the layout. Some of these images served as links that sent commands to the 3D world, and some of these images were controlled by click events from within the 3D world. The "region" system (borrowed from SMIL) provides a layout system for the puppeting of sprites to display such images and text. The largest challenge in adding this functionality would be the formatting of text objects, and the methodology for including that formatting within the XML document.

- **Multiple Worlds & Scenes**

  At this time, XDRIVE presentations only support one W3D file per presentation. Future development should include the capability to load multiple 3D files, import objects from other files, and jump back and forth between 3D worlds, preferably remembering and restoring the state of a 3D file when a presentation moves to another file then moves back. Scenes could have durations, and commands to jump to other scenes / worlds could be included as a default at the end of a scene.

  The importation of models and other resources from an external file that could serve as a resource library could facilitate the display of multiple instances of the same model, or the inclusion only of necessary models, useful for worlds with large amounts of information where only a portion of that information needs to be active within the system at a given time.

- **Layout Properties**

  Controlling the size of the Shockwave object within the browser and controlling the size of the movie's stage would allow for multiple orientations instead of the default 640 x 480 size currently used. Additional properties could control other elements of the layout, such assigning a color or an image as the background of a presentation. Connecting properties to the directional and ambient lighting systems in Shockwave3D to allow for additional control over the appearance of the 3D world. Also, camera overlays and backdrops – images that appear in either the background or over top of the world – would add an additional level of control over the user experience.

- **Navigation Changes**

  In its current version, the XDRIVE system only allows for control over the camera system at the settings level. Commands could be added to automate changes to the navigation system, such as turning user-led navigation on and off. New navigational / interaction commands could be added, or custom key buttons mapped to call triggers within the system. Additional navigation styles could be introduced as well, such as restricting motion of the camera to only orbit an object, or allowing click-drag rotation or movement of objects within the scene.

- **Image Texturing**

  Although images mapped to models are imported from the W3D, XDRIVE currently provides no control over the display of those images as textures. Additional features could be added allowing greater control over those images, such as the importation of other images and mapping and positioning controls for those images, and alpha mapping channels for those images. In the future, Shockwave3D may also be able to support additional texture technologies such as bump mapping and normal mapping.

- **Collision Detection and Boundary Systems**

  Collision detection functions could be added to the system that would restrict motion of the camera object when it encounters a surface. These systems can be complex, so the most likely method for implementing this feature would be to create a convex-hull system that would create simple shapes around complex models and tests the camera object's position against those shapes. Boundary systems could also be included that would restrict the overall motion of the camera object, either through the definition of a shape (simple box or sphere) or through the use of a custom shape – the camera object would test all camera navigation events to ensure that the camera object remained within the set boundaries. The proximity detection system could also be altered to define custom volumes that would result in a call to trigger events, or to support the incremental evaluation of a trigger, such as an object who's opacity level is dependent on the distance to the camera object.

- **Additional Error Checking / Path Verification**

  Currently error-checking and reporting is minimal at best, and the author experience could be enhanced through the inclusion of diagnostic and verification systems that could report back during the authoring process. A central method of file and path verifications, as well as a method for replacing or ignoring unloaded objects would also benefit developers.

- **Audiovisual Objects**

  Additional development might support additional AV objects and the ability to control the properties of those objects through the command language. Playback commands could be sent along the timeline, and object properties such as volume could be controlled. Also, other player formats such as Quicktime or Windows Media could be included.

- **Built-in Animations**

  Shockwave3D objects can have animations embedded within them, and accessed at the script level. These animations can be created through coding or through a keyframing process during model export from other systems such as 3D Studio MAX. The ability to access and control these animations, or to create these animations components could allow for more complex animations to be included and referenced in the command file only by name.

- **Repeating Animations & Actions**

  Animations that cycle a set number of times or indefinitely would allow certain actions to persist without a great level of control. Multiple actions or animations stored as named instance could then be assigned to objects at different stages, and switched accordingly.

- **Server Technologies and Multi-user Environments**

  Director has the capability to send information to and receive information from remote machines through various sockets. The system could be enhanced to send input back to a central server through POST commands, useful in situations where it would be desirable to capture user events, such as online testing or retail. This could also be altered to download and display custom information, such as GIS data. Through the use of XML sockets, the system could be revised to send and receive live data via XML, allowing for presentations to be controlled from a central server, or from a link with other client machines, creating multi-user environments.

14.0 POTENTIAL USES FOR XDRIVE

- **Architectural Walkthroughs**

  One of XDRIVE's most obvious potential uses is as a system to create virtual walkthroughs of architectural projects with advanced features. Architects could narrate the project as the user is guided through a project, or users could freely explore the project. Clients could interact with the project, possibly experiencing the same space with different options or design elements, allowing them to make a more informed choice and provide an instant comparison. Architectural history courses could provide 3D versions of subject buildings allowing students to explore the great works as spatial creations rather than images and abstractions, and to interact with information embedded within the model.

- **Online education**

  Architecture is not the only field that could benefit from utilizing virtual environments as educational components. Any field where spatial exploration could enhance the quality of instruction could employ this system to demonstrate complex structures, and could employ the additional media to create an encompassing instructional component. Some of these fields may include chemistry, engineering, urban planning, military training, and medical sciences. These fields are also not limited to a particular educational level - presentations could be built to assist in teaching physics for high school students, or in basic spatial relations and directions for elementary school children. Of course, the inclusion of 3D content need not be necessary for a lesson – it could also be included just for the "wow" factor.

- **Product sales**

  Next to games, product sales are making the most use of web-based technologies to enhance the buyer's experience and education. The XDRIVE system could be used to display 3D representations products to potential buyers, and even let those buyers interact with the product, such as changing the options on a vehicle, or the color of the vehicle itself.

  Another potential product market could be enhancing event ticket sales. Rather than having a buyer select tickets based on a map of the venue, s/he could ascertain the quality of the seats by experiencing them – viewing a model of the venue from the position of the seat they are considering purchasing. This can be taken a step further – by adding connectivity to a database, the buyer could see what other seats are available through some indicator such as color or highlighting.

- **Wayfinding**

Similar to architectural walkthroughs but customized to the task at hand, the system could be used as a wayfinding tool, providing users with a spatial representation of a travel path, rather than relying on text directions or plans. Through the use of CGI, a path could be determined through the evaluation of the shortest node-to-node path (similar to Mapquest and other systems), and then converted into an animation path for playback in the XDRIVE system. This style of landmark-based navigation may prove to be especially effective in large or complex environments such as hospitals, university campuses, or industrial plants.

- **Retail**

  In one existing use of the XDRIVE system, Katrina Jones Choi created "KAJO", a virtual grocery store, for her master's thesis. In this concept, users navigated through a three-dimensional representation of a grocery store, selecting items, viewing 3D models of those items as well as prices and nutritional information, and adding them to their "cart". This way users still had a grocery store retail experience, without the brick-and-mortar store – the online retailer would simply fill the order and deliver it to the user. Product Sales have already been covered, but this method could also be used for retailers as well to view multiple projects. It could also be used by retail complexes such as malls and urban centers as a method for users to find information about a particular business – users could navigate a virtual complex the way they would the real complex, and interact with the individual business, perhaps taking them to the company website, or allowing the company to display certain items in the "window".

- **Information Visualization**

  The system could be altered to allow for the creation and alteration of model mesh resources and advanced texturing to allow for information visualization systems to take advantage of the 3D capabilities of this system, such as complex graphs or mathematical forms.

- **Event Playback**

  A pre-recorded series of events could be recreated using the XDRIVE system, assisting in the advanced analysis of an event by allowing spatial exploration of it. While I cannot begin to guess at the number of possible events people may want to recreate, some examples of this might be: accident and crime scene recreation, military operation analysis, and egress simulations.

- **Games**
  Through the use of actions or the addition of connections to external control commands such as Javascript, games may be built using this system. These games could be for educational purposes, or even just for fun.

15.0 CONCLUSION

In undertaking this thesis project, I set out to create a system that would simplify the development process involved in creating real-time three-dimensional environments to use for online instruction and communication. This system was designed to provide presentation authors with a simple and efficient interface with which to define and control their creations. This was accomplished through the programming of function systems that coordinate and control the internal operations required to generate these presentations. It is not my intention to define what developers should or should not create, rather it is my hope that this system can open these technologies to a wider audience, allowing them to bring their own imagination to the process.

RESOURCES USED

Catanese, Paul
Director's Third Dimension
Fundamentals of 3D Programming in Director 8.5
Indianapolis, Indiana: QUE Publishers, 2002

Parent, Rick
Computer Animation
Algorithms and Techniques
San Francisco, California: Morgan Kaufmann Publishers, 2002

Macromedia
Director MX Help Files

APPENDIX A: XDRIVE XML Language Definition

The following describes the structure to which an XML document must adhere to be considered readable by the XDRIVE engine, as well as a definition of all valid tags and a description of the purpose and correct application of each, as well as the attribute names and appropriate values and/or value types associated with each tag.

These definitions and descriptions conform to that which is readable by the engine at the time of publication and defense.

XDRIVE:: XML STRUCTURE – WORKING DRAFT v. 1.0
6-30-06

```
<?xml version=”1.0”?> (0.0)
<xdr version=”1.0”>  (0.1)
     <setup> (1.0)
          <layout> (2.0)
               <region /> (2.1)
          </layout>
          <preferences> (3.0)
               <navigation /> (3.1)
               <keymap /> (3.2)
               <speed /> (3.3)
          </preferences>
          <worldlist> (4.0)
               <world /> (4.1)
          </worldlist>
     </setup>
     <body> (5.0)
          <worldsettings>
               <position />
               <rotation />
               <scale />
               <highlight />
               <camera />
               <headlight />
               <fading />
          </worldsettings>
          <preworld>  (5.1)
               …
          </preworld>
          <scene> (6.0)
               <prescene> (6.1)
                    …
               </prescene>
          </scene>
     </body>
</xdr>


<camerafade />
```

XML Document Declaration

<?xml version="1.0"?>

Declares the XML format & character-set. Required by Director XML Parser to correctly interpret XML data

0.0 Setup Tags

<setup> … </setup>

Defines the "setup" area of the file. Contains the scene layout, navigation settings and user preferences for the presentation

1.0 Layout Tags

<layout>…</layout>

Area to define the media region layout for the presentation.

1.1 Region Tag

<region *[attributes]* />
name="region id"
top: "x-value"
left: "y-value"
width: "width-value"
height: "height-value"
z="depth" *future feature – not yet active*

Defines an area of the screen to hold media. 3D, image, and text objects will refer to their region for where their placement should be. (Similar to SMIL)

2.0 Preferences Tags

<preferences> … </preferences>

Holds the default preferences for navigation, keymap & speed settings

2.1 Navigation Tag

<navigation *[attributes]* />
enabled: TRUE or FALSE
type: free or ??
mouse_enabled: TRUE or FALSE
mouse_buffer: %
restrict_zaxis: TRUE or FALSE
boundary: TRUE or FALSE
proximity: TRUE or FALSE
collision: TRUE or FALSE

Adjusts the default navigation settings for all scenes called from this file.

Future Development – Navigation settings will be adjustable from within individual scenes.

2.2 Keymap Tag

<keymap *[attributes]* />
movefoward: charnum
movebackward: charnum
moveleft: charnum
moveright: charnum
turnup: charnum
turndown: charnum
turnleft: charnum
turnright: charnum
mouselook: charnum

Defines adjustments to the default keymapping. This map will be used as the default for all scenes within the file.

2.3 Speed Tag

<speed *[attributes]* />
keymove: unit float
keyrotate: degree float
mouserotate: degree float (max value)
invertmouse: TRUE or FALSE

Defines adjustments to the default speed settings. These settings will be used as the default for all scenes within the file.

3.0 Worldlist Tags

<worldlist> … </worldlist>

Holds the list of W3D files that will be called by this file.

3.1 World Tag

<world *[attributes]* />
name: world id
path: W3D filepath

Pairs a name identifier with a W3D filepath. Scenes will associate with this world through the “worldname” attribute.

4.0 AV List Tags

<avlist> … </avlist>

Holds the list of audio / video objects that can be used as a timeline by the scene

4.1 AV-Object Tag

<av-object *[attributes]* />
name: unique name

path: av filepath
type: REALPLAYER, QUICKTIME or WINDOWSMEDIA

Assigns a name identifier to the file path and media type of an av object. The scene tag will reference this tag if needed as an external timeline synchronization object. Currently, only files formatted in RealPlayer format are supported.

5.0 Body Tags

<body *[attributes]*> … </body>
world: world id
region: region id

5.1 Worldsettings Tags

<worldsettings *[attributes]*> … </worldsettings>
inherit: TRUE or FALSE
background: hexval
ambient: hexval

Contains the overall settings for the current world (position, rotation, lighting, etc)

5.1.1 Position Tag (optional)

<position *[attributes]* />
x: x-position
y: y-position
z: z-position

Sets the XYZ coordinates for the initial model position

5.1.2 Rotation Tag (optional)

<rotation *[attributes]* />
x: x-rotation degrees float
y: y-rotation degrees float
z: z-rotation degrees float

Alters the XYZ rotation for the initial model position

5.1.3 Scale Tag (optional)

<scale *[attributes]* />
x: x-scale percent float
y: y-scale percent float
z: z-scale percent float

Alters the XYZ scale for the entire model

5.1.4 Highlight Tag (optional)

<highlight *[attributes]* />
time: cycle time ms
color: hexvalue
ambient: hexvalue

Alters the default settings for the highlight system

5.1.5 Camera Tag (optional)

<camera *[attributes]* />
x: x-coord float
y: y-coord float
z: z-coord float
pan: pan angle float
tilt: tilt angle float (-90 – 90)
fadecolor: hexval

Sets the initial camera position and settings for the world.

5.1.6 Headlight Tag (optional)

<headlight *[attributes]* />
enabled: TRUE or FALSE
type: “spot” or “directional”
color: hexvalue

6.0 Setup Command Tags

These are one-time commands, with no duration value. They can have a start time, but have a duration of 0.

6.1 Disinclude Tag

<disinclude *[attributes]* />
object: object name
type: object type (optional)

Removes this object completely from the world. Cannot be turned back on. If type is not set, it will search for a matching object name (order: model, group, light)

6.2 Group Tags

<group *[attributes]* > … </group>
name: group name

Takes a list of <object> tags and puts the models & lights, into a group (maybe include groups later, time permitting)

6.3 Ungroup Tag

<ungroup *[attributes]* />
name: group name

Destroys the grouping

7.0 Scene Tags

<scene *[attributes]*> … </scene>
name: unique scene name
avobject: av object name
avregion: region for av playback

The scene tags enclose an entire scene. Scenes have a name, and if they have an "avobject" value, that named av_object will serve as the external timeline for the scene. All actions are contained within the <scene> tags.

8.0 Queue Command Tags

All following tags share the following attributes:
name: unique id (optional)
start: time (ms), NOW, or HOLD (disregarded in preworld & prescene)
duration: time (ms) (disregarded in preworld & prescene)
holdfor: unique id (optional)
verify: TRUE or FALSE (optional. Default: FALSE)

8.1 Camerafade Tag

<camerafade *[attributes]* />
direction: IN or OUT

Camera Fade is a full fade in or out. (100% - future development may include a partial fade)

8.2 Toggle Tag

<toggle *[attributes]* />
object: object name
type: MODEL, GROUP or LIGHT (optional)
direction: ON or OFF

Turns the specified model or light on or off.

8.3 Lightcolor Tag

<lightcolor *[attributes]* />
light: light name or BACKGROUND
color: hexval

Sets the corresponding light to the specified color. If duration is > 0, light change is animated through RGB interpolation.

8.4 Setcolor Tag

<setcolor *[attributes]* />

object: object name or BACKGROUND
type: MODEL, GROUP, or LIGHT
color: hexval

Sets the corresponding object to the color. If model or group, runs the alternate shader. If light, runs the lightcolor command. If name = “background” and type=”light”, sets the background color. Durations > 0 will animate the change through RGB interpolation.

8.5 Resetcolor Tag

<resetcolor *[attributes]* />
object: object name
type: MODEL or GROUP

Sets the corresponding object back to it’s original shader.

8.6 Setopacity Tag

<setopacity *[attributes]* />
object: object name
type: MODEL or GROUP
percent: opacity percentage value (0=clear, 100=opaque)

Sets the corresponding model or group object’s opacity to the percentage value. If duration value > 0, opacity change will animate through interpolation.

8.7 Highlight Object Tag

<highlight-object *[attributes]*/>
object: object name
duration: time (ms) or INDEFINITE

Highlights the specified group name or object. A start time will queue the event. A set duration will also register an unhighlight-object event.

8.8 UnHighlight-Object Tag

<unhighlight-object *[attributes]*/>
object: object name

Resets the highlight of the named group or model. Start time will queue the event.

8.9 Point Camera / Object At tags:

Common Attributes:
accel: time (ms)
decel: time (ms)
persist: TRUE or FALSE
persist-for: time (ms)

Object (non-camera) Attributes:
object: object id

type: object type (NOT CURRENTLY FUNCTIONAL)

Rotates the camera or specified object to point (face) towards the appropriate location or direction.   Durations > 0 will result in the animated rotation of the camera object. Accel & Decel times are used to control ease in/out curves.

CAMERA & OBJECT "Points" (Targeting) do not have the following attributes: "name", "holdfor", "verify". HOLD is not a valid start-time entry)

8.9.1 Point Camera At Location

<point-camera-at-location />
x: x-position
y: y-position
z: z-position

8.9.2 Point Camera At Object

<point-camera-at-object />
target-object: object name

8.9.3 Point Camera At Direction

<point-camera-at-direction />
x: x-position
y: y-position
z: z-position

XYZ point is relative to camera location. Resolves to a vector direction within Director.

8.9.4 PointCameraAtPath

<point-camera-at-path />
no specific attributes required

If the camera is following a path, will use the path direction as the direction to face.

8.9.5 PointCameraAtNone

<point-camera-at-none />
no specific attributes required

Cancels any current persistent target for the camera. Takes no duration.

8.9.6 Point-Object-At-Location

<point-object-at-location *[attributes]*/>
x: x-position

y: y-position
z: z-position

Points the object facing this specified location.

8.9.7 Point-Object-At-Direction

<point-object-at-direction *[attributes]* />
x: x-position
y: y-position
z: z-position

Points this object in the specified direction.

8.9.8 Point-Object-At-Object

<point-object-at-object *[attributes]* />
target-object: object name

Points this object to face the target-object

8.9.9 Point Object At Path

<point-object-at-path *[attributes]* />
no type specific attributes

If object is in motion, sets the object to face the path direction.

8.9.10 Point Object At None

<point-object-at-none *[attributes]* />
no type specific attributes

Cancels any persistent target holds on the named object.

9.0 Motion Tags

<camera-motion *[attributes]* >
{sequence of motions}
</camera-motion>

<object-motion *[attributes]* >
{sequence of motions}
</object-motion>

accel: acceleration time (ms)
decel: deceleration time (ms)
start-position: POSITION or HERE
x: x-position (for POSITION only)
y: y-position (for POSITION only)
z: z-position (for POSITION only)
object: object name (object-motion only)
type: object type (object-motion only)

Sets a motion path for the camera or specified object. Within the <motion></motion> tags, a list of specific motions with position data guide the total motion path. Attributes for the <motion> tags are similar to queue event command attributes (name, start, duration, holdfor, verify)

9.1 Direct Motion Tag

<direct *[attributes]* />
x: x-position
y: y-position
z: z-position

Direct linear motion to an absolute point (world coordinates).

9.2 Relative Motion Tag

<relative *[attributes]* />
x: x-position
y: y-position
z: z-position

Direct linear motion to a point relative. (Offset)

9.3 Orbit Motion Tags

<orbit *[attributes]* > … </orbit>
angle: angle of orbit rotation (float)
pivot: OBJECT or POINT
pivot-object: object name (pivot = OBJECT only)
axis: X-AXIS, Y-AXIS, Z-AXIS or CUSTOM
rotate: TRUE or FALSE

Camera or Object orbits the specified pivot point by a specified number of degrees around the specified axis. CUSTOM value for "pivot" or "angle" will check sub object (xml resource) for additional data, or default to 0,0,0, z-axis. Rotate attribute determines whether the object will hold a fixed orientation, or turn as the angle turns.

9.3.1 Pivot Tag

<pivot *[attributes]* />
x: x-position
y: y-position
z: z-position

Specifies a world coordinate pivot point.

9.3.2 Axis Tag

<axis *[attributes]* />
x: x-value
y: y-value
z: z-value

XYZ Coordinates specify an axis vector to rotate object around.

9.4 Path Motion Tags

<path *[attributes]* > … </path>
no type specific attributes

Defines a spline path through a nested sequence of points.

9.4.1 Point Tag

<point *[attributes]* />
x: x-position
y: y-position
z: z-position

Defines a spline checkpoint.

10.0 Rotation Tags

These can be called during a scene, or as a prescene. They are a relative rotation for the object or camera. Durations > 0 during a scene will result in an animation. They share the same regular attributes as all queue commands (name, start, duration, hold-for) as well as motion tags (accel, decel)

10.1 Rotate Object

<rotate-object *[attributes]*/>
object: object name
type: object type
reference: WORLD or SELF (world is default)
angle: angle of rotation (float)
axis: X-AXIS, Y-AXIS, Z-AXIS or CUSTOM (custom uses <axis> tag)

Rotates an object around it's own pivot point by the specified angle. Rotation axis is determined relative to either the world axis or the objects self axis. An axis value of CUSTOM will require the use of the nested <axis/> tag.

10.2 <adjust-camera *[attributes]*/>

pan: pan angle (float)
tilt: tilt-angle (float)

Rotates the camera's angle relative to the provided attributes

10.3 <rotate-camera *[attributes]*/>

angle: angle of rotation (float)
axis: X-AXIS, Y-AXIS, Z-AXIS, or CUSTOM

Rotates the camera about it's central point by the specified angle. Angle is always relative to the current camera object. An axis value of CUSTOM will require the use of the nested <axis/> tag.

10.4 **<set-object-orientation *[attributes]*/>**

object: object-name
type: object-type
angle: angle of rotation (float)
axis: X-AXIS, Y-AXIS, Z-AXIS or CUSTOM

Rotates the point-at orientation of an object about the specified axis through it's pivot point by the specified angle. An axis value of CUSTOM will require the use of the nexted <axis/> tag.

10.5 **<reset-object-orientation *[attributes]*/>**

object: object-name
type: object-type (optional)

Resets an objects orientation to it's original value.

11.0 Position & Direction Tags

These can be called during a scene, or as a prescene. They are a discrete position set with no duration.

11.1 Set Position

<set-position *[attributes]*/>
object: object name
type: object type
x: x-position
y: y-position
z: z-position
start: start time (ms) {where applicable}

Sets the position of the named object to the specified XYZ coordinate. All coordinates must be provided for the positioning to be valid. If no start time is provided, starttime will be assumed as 0.

11.2 Set Camera Position

<set-camera-position *[attributes]* />
x: x-position
y: y-position
z: z-position
start: start time (ms) {where applicable}

12.0 Trigger Commands

Trigger commands are used to set and reset model-click, model-proximity, and named triggers (future development)

12.1 Trigger Tags

<trigger *[attributes]*> … </trigger>
name: unique trigger name

The trigger tag allows the user to define a series of nested commands as a named trigger. Triggers can be called by name from click & proximity events.

12.2 Set Click Trigger

<set-click-trigger *[attributes]*> … </set-click-trigger>
OR
<set-click-trigger *[attributes]*/>
start: start time (ms)
duration: duration time (ms) or INDEFINITE
object: object name
continue: TRUE or FALSE
trigger: trigger name (optional)

The Set Click Trigger Tags registers the contained tags as a click event for a model. (model only – you can't click a group, light or camera). If a start time is set, the registration only occurs once that time has past. The duration for this tag defaults to INDEFINITE, meaning no expiration. If a time is set, the model click event will unregister after that amount of time. A continue value of FALSE will unregister the click event after a click event has occurred. TRUE will result in the click event remaining active and potentially being called again. Choosing a named trigger using the "trigger" attribute will associate this click event with that trigger.

12.3 Reset Click Trigger

<reset-click-trigger *[attributes]*/>
start: start time (ms)
object: object name

Kills any registered model click event in the model.

12.4 Set Proximity Trigger

<set-proximity-trigger *[attributes]*/>
start: start time (ms)
duration: time (ms) or INDEFINITE
object: object name
distance: units (float)
trigger: trigger name
leave-trigger: trigger name or NONE
continue: TRUE or FALSE

Sets a proximity trigger for the named object. When the camera is within a set distance of the object, then the named trigger will be called. If a "leave-trigger" is set, that named trigger will be called when the camera is once again outside of that distance. Proximities become active at the start time (default: 0), and run throughout the set duration (default: indefinite). The "continue" attribute sets the persistence of this trigger. A "FALSE" value will result in the trigger expiring after one occurence. A "TRUE" value will result in repeated calls to the triggers each time the event is reached. Models may only have one proximity trigger.

12.5 Reset Proximity Trigger
<reset-proximity-trigger *[attributes]*/>

start: start time (ms)
object: name

Deactivates the proximity trigger for this object.

13.0 Other Commands

13.1 Web Link Tag

<web-link *[attributes]*/>
href: URL resource
window: target window name

Opens a web page to the specified URL. The “window” attribute is optional. If the name value for the “window” attribute corresponds with the name of an open browser window, that window will redirect to the new URL. If no window holds that name, a new browser window with that name will open. A value of “_new” will open a new browser window. An empty value (default) will redirect the current browser window the presentation is playing in.
NOTE: This does not have a starttime / duration / name attribute. Use this with triggers, or as the redirect for a scene.

***: Disregarded in Pre-world & Pre-scene environments**

APPENDIX B: XDRIVE Script List

The following content is comprised of the uncompressed scripts contained within the XDRIVE engine. They are listed in the order that they appear in the Internal Cast in the Director File.

These scripts are those that form the engine at the time of publication and defense.

Engine Setup

```
-- 3DSYSTEM SETUP SCRIPTS --
-- Scripts began: 01/01/03 --
--
-- "scene" object references the 3dDisplayModel member
-- sceneLayoutProps holds the values for the scene object
layout

global scene
global sceneLayoutProps
global scenemodels
global currentscene
global user_keymap
global user_speedsettings
global user_restrictions
global current_model_list
global current_group_list
global current_light_list
global invisible_model_list
global off_light_list
global default_fade_image
global custom_shader_list
global model_highlight_list
global model_opacity_list
global model_color_list
global model_orientation_list
global group_orientation_list
global light_orientation_list
global camera_orientation
global object_target_list
global av_object_list
global av_object
global region_map
global topsprite
global xml_doc
global XML_member
global XML_worldsettings
global XML_preworld
global XML_scenelist
global diagnostic
global current_scene_id
global base_scene_id
global motion_catalog
global click_trigger_list
global trigger_xml_list
global prox_trigger_list
global current_prox_on
global current_prox_off
```

```
-- catmull rom globals
global cr_M
global cr_MBx
global cr_MBy
global cr_MBz


on preparemovie
  -- set the initial variable & system queuing

  sendStatus("Initializing engine")

  -- associate the "scene" item
  scene = member("3dDisplayModel")

  -- create the default layout properties for the 3D element
  sceneLayoutProps = [#x:0, #y:0, #w:0, #h:0, #spritenum:0]

  -- create the scenemodels list
  scenemodels = []

  -- create the current model / group / light list
  current_model_list = []
  current_group_list = []
  current_light_list = []
  model_highlight_list = [:]
  model_opacity_list = [:]
  model_color_list = [:]
  invisible_model_list = [:]
  off_light_list = [:]
  custom_shader_list = [:]

  model_orientation_list = [:]
  group_orientation_list = [:]
  light_orientation_list = [:]
  camera_orientation = 0
  object_target_list = [:]

  click_trigger_list = [:]
  trigger_xml_list = [:]

  prox_trigger_list = [:]
  current_prox_on = []
  current_prox_off = []

  -- reset the catmull rom globals
  cr_M = VOID
  cr_MBx = VOID
  cr_MBy = VOID
  cr_MBz = VOID
  -- set the catmull-rom M matrix
  CR_MSetup()


  -- create the camera fade images
  default_fade_image = VOID

  -- create the av object list
  av_object_list = []
  av_object = VOID

  -- create the region_map & topsprite object
  region_map = [:]
  topsprite = 11

  -- create the motion catalog item
  motion_catalog = [:]

  -- create the XML holding document
  xml_doc = VOID
  xml_worldsettings = VOID
  xml_preworld = VOID
  xml_scenelist = VOID


  -- create the diagnostic object
  diagnostic = FALSE

  -- startTemporarySettings()

  -- setup the default settings for the player
  buildDefaultSettings()

  -- load the XML file
  loadXML()

  -- set the current scene id to the 1st scene object
(Default)
  current_scene_id = VOID
  base_scene_id = VOID



end


on stopmovie

  -- erase any leftovers in model
  scene.resetworld()

  -- kill the XML document member
  erase(XML_member)
  clearglobals
  the actorlist = []


end




on buildDefaultSettings
  -- Builds the default structure / settings for the player.
Later,
  -- the XML file will replace defaults with specific
settings

  sendStatus("Building default settings")

  -- build the currentscene object
  currentscene = [#scenename:"", #settings:[:]]

  -- TIMELINE SETTINGS hold the settings for the timeline
  -- #starttime: starting milliseconds value (greater than
zero indicates returning to a scene)
  -- #type: #internal or #external (external denotes
timeline is A/V dependent)
  -- #attributes:  Holds the attributes for the A/V
dependant file (#external timeline only)
  --      #name: <av object list ID>. AV File Location
(streaming / local)
  --      #start_time: <time>.   What location within the
file to start from
  --      #duration:  <time>.  Duration of this clip, or
#all (plays the entire duration)
  defaultTimelineSettings = [#starttime:0, #type: #internal,
#attributes:[]]

  -- #NAVIGATION SETTINGS hold the nav settings for the
scene
  -- #enabled: TRUE or FALSE.  Is user-navigation enabled.
  -- #type: #free, ?, or ?.  User-navigation style (free
denotes flythrough)
  -- #mouse: Mouse Navigational Settings
  --      #enabled: TRUE or FALSE.  Is the mouse-look
feature enabled
  --      #buffer_percent: <num>.  Percentage-of-sprite area
(centered) where mouse-look will not occur.
  -- #restrictions:
  --      #zaxis: TRUE or FALSE.  Can the user fly-up and
down in free-form?
  -- #boundary: Model Boundary Settings
  --      #enabled: TRUE or FALSE.  Is the boundary
condition setting on?
  -- #collision: Model Collision Settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the collision setting
on?
  -- #position: Position determination settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the position-event
feature on?
  defaultNavigationSettings = [#enabled: TRUE, #type: #free,
#restrictions:[#zaxis: FALSE], #mouse:[#enabled: TRUE,
#buffer_percent:10], #boundary:[#enabled:FALSE],
#collision:[#enabled:FALSE], #position:[#enabled:FALSE]]



  -- #MODEL SETTINGS holds the settings for the current
model
  -- #worldsettings:
  --      #structure:
  --            #preserve_structure: TRUE or FALSE.
Preserve inherent parent / child relationships
  --      #orientation:
  --            #position: <vector>  pre-positioning of the
model
  --            #rotation: <vector>  pre-rotation of the
model
  --            #scale: <vector>  pre-scale of the model
  --      #highlight:  SETTINGS FOR MODEL HIGHLIGHT
  --            #cycle_duration: <milliseconds>.  Determines
the cycle time for this event
  --            #diffuse: <rgbvalue>.  Maximum diffuse color
of highlight shader
  --            #ambient: <rgbvalue>.  Maximum ambient color
of highlight shader
  -- #groups:  Pre-defined groups by name & members (models
& lights)

  -- worldsettings
  -- NOTE: THE DEFAULT ROTATION IS X:-90 TO COUNTER VIZ
EXPORT AXIS ISSUES (Y = Z, Z = Y)
  defaultModelOrientation = [#position: vector(0,0,0),
#rotation: vector(-90,0,0), #scale: vector(1,1,1)]

  -- build default worldsettings
  defaultWorldSettings = [#structure: [#preserve_structure:
FALSE], #highlight:[#cycle_duration:3000, #diffuse:
rgb(255,255,100), #ambient: rgb(0,0,50)], #orientation:
defaultModelOrientation]

  -- build default model settings
  defaultModelSettings = [#worldsettings:
defaultWorldSettings]


  -- #LIGHTING SETTINGS holds the light settings for the
current model
  -- #worldsettings:
```

```
  --      #background_color: <hexvalue>
  --      #ambient_color: <hexvalue>
  --      #directional: ??? INCLUDE THIS ???
  defaultLightingSettings =
[#worldsettings:[#background_color: "#000000",
#ambient_color:"#000000"]]


  -- #CAMERA SETTINGS holds the settings for the current
scene's camera
  -- #initial_position: holds the translation for the
initial orientation of the camera
  --       #x_pos: <float>  The x position of the camera in
units
  --       #y_pos: <float>  The y position of the camera in
units
  --       #z_pos: <float>  The z position of the camera in
units
  --       #pan_deg: <float>  The degrees to pan the camera
  --       #tilt_deg: <float>  The degrees to tilt the
camera
  -- #headlight:  holds the properties for the camera
headlight object
  --       #enabled: TRUE or FALSE.  Is the headlight
enabled?
  --       #type: #spot or #directional.  Type of light
being used.
  --       #color: <hexval>.  Color / Intensity of the
headlight.
  -- #fading:  holds the properties for the camera fades
  --       #default_color: <hexval>.  Default fade color
value
  --       #custom: <rgbcolor>.  List of members for "custom
fades"
  defaultInitialCameraPosition = [#x_pos: 0, #y_pos: 0,
#z_pos: 0, #pan_deg:0, #tilt_deg: 0]
  defaultCameraFading = [#default_color: "#000000",
#custom:[]]
  defaultCameraSettings = [#initial_position:
defaultInitialCameraPosition, #headlight: [#enabled: FALSE,
#type:#spot, #color:rgb(200,200,200)],
#fading:defaultCameraFading]

  -- fill the currentscene settings
  currentscene.settings = [#timeline:
defaultTimelineSettings, #navigation:
defaultNavigationSettings, #model: defaultModelSettings,
#lighting: defaultLightingSettings, #camera:
defaultCameraSettings]


  -- TEMPORARY NAVIGATIONAL SETTINGS
  user_keymap = [#move_forward: 30, #move_backward:31,
#move_left:28, #move_right:29, #rotate_up:119,
#rotate_down:115, #rotate_left:97, #rotate_right:100,
#mouse_look:109]
  user_speedsettings = [#move: 10, #rotate:1,
#mouse_speed:5, #mouse_invert:FALSE]




  -- 3D object setup - center, with 50px margin
  sceneLayoutProps.x = 50
  sceneLayoutProps.y = 50
  sceneLayoutProps.w = 540
  sceneLayoutProps.h = 380

end


on startTemporarySettings

  -- update the status bar
  sendStatus("Building temporary settings")

  -- temporary housing for initial settings, until they can
be
  -- imported from external sources

  sceneLayoutProps.x = 50
  sceneLayoutProps.y = 50
  sceneLayoutProps.w = 540
  sceneLayoutProps.h = 380

  scenemodels = [[#name: "gothic", #path:
"navetest3.W3D"],[#name:"kajo", #path: "/apple.w3d"],
[#name:"denis", #path: "/st_denis.w3d"]]

  -- currentscene = [#scenenum: 1, #scenename: "gothic",
#settings:[#starttime: 0, #type: #internal,
#navigation:[#enabled:TRUE, #type:#free],
#mouse:[#enabled:TRUE, #buffer_percent:10], #position:FALSE,
#attributes:[:]]]

  -- set up the current scene (temporary population until
parameters are
  -- imported from the document or loaded by default

  -- currentscene = [#scenenum: 1, #scenename: "gothic",
#settings:[:]]
  currentscene = [#scenename: "denis", #settings:[:]]

  -- LAYOUT SETTINGS hold the settings for the screen layout
  -- Layout types:
  --      #3D
  --      #audio
  --      #video
  --      #image
  --      #text


  -- MEDIA OBJECTS
  -- Setting a temporary av object as a name for the
timeline to recognize & load
  -- "tempAVSettings" holds the av object attributes
  --     #name:  The media object name that this will
recognize
  --     #attributes:
  --      #filepath:  The location of the file to load
  --      #mediatype:  #realplayer, #quicktime OR #windows.
Currently only Real is available
  --      #avtype: #audio OR #video.  Use this to determine
the visibility of the videobox sprite
  --      #location:  [#x: xloc, #y: yloc, #w: width, #h:
height]

  av_object_list = [[#name:"postal", #attributes:[#filepath:
"track2.rm", #mediatype: #realplayer,
#size:[#x:0,#y:0,#w:0,#h:0]]]]




  -- TIMELINE SETTINGS hold the settings for the timeline
  -- #starttime: starting milliseconds value (greater than
zero indicates returning to a scene)
  -- #type: #internal or #external (external denotes
timeline is A/V dependent)
  -- #attributes:  Holds the attributes for the A/V
dependant file (#external timeline only)
  --      #name: <av object list ID>. AV File Location
(streaming / local)
  --      #start_time: <time>.   What location within the
file to start from
  --      #duration:  <time>.  Duration of this clip, or
#all (plays the entire duration)

  -- SAMPLE TIMELINE SETTINGS - NON-AV DEPENDENT
  -- tempTimelineSettings = [#starttime: 0, #type:
#internal, #attributes:[]]

  -- SAMPLE TIMELINE SETTINGS - AV DEPENDENT
  -- REGULAR TEST SETTINGS
  -- tempTimelineSettings = [#starttime: 0, #type:
#external, #attributes:[#name: "postal", #start_time: 0,
#duration: #all]]

  -- ST. DENIS SETTINGS
  tempTimelineSettings = [#starttime:0, #type: #internal,
#attributes:[]]
```

```
  -- #NAVIGATION SETTINGS hold the nav settings for the
scene
  -- #enabled: TRUE or FALSE.  Is user-navigation enabled.
  -- #type: #free, ?, or ?.  User-navigation style (free
denotes flythrough)
  -- #mouse: Mouse Navigational Settings
  --      #enabled: TRUE or FALSE.  Is the mouse-look
feature enabled
  --      #buffer_percent: <num>.  Percentage-of-sprite area
(centered) where mouse-look will not occur.
  -- #restrictions:
  --      #zaxis: TRUE or FALSE.  Can the user fly-up and
down in free-form?
  -- #boundary: Model Boundary Settings
  --      #enabled: TRUE or FALSE.  Is the boundary
condition setting on?
  -- #collision: Model Collision Settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the collision setting
on?
  -- #position: Position determination settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the position-event
feature on?

  -- SAMPLE TEMP NAV SETTINGS - FREE EXPLORE
  tempNavigationSettings = [#enabled: TRUE, #type: #free,
#restrictions:[#zaxis: FALSE], #mouse:[#enabled: TRUE,
#buffer_percent:10], #boundary:[#enabled:FALSE],
#collision:[#enabled:FALSE], #position:[#enabled:FALSE]]

  -- SAMPLE TEMP NAV SETTINGS - SLAVE EXPLORE
  --  tempNavigationSettings = [#enabled: FALSE, #type:
#free, #restrictions:[#zaxis: FALSE], #mouse:[#enabled:
TRUE, #buffer_percent:10], #boundary:[#enabled:FALSE],
#collision:[#enabled:FALSE], #position:[#enabled:FALSE]]


  -- #MODEL SETTINGS holds the settings for the current
model
  -- #path: <dirpath>.  ???? SHOULD I EVEN DO THIS ?????
  -- #worldsettings:
  --      #structure:
  --            #preserve_structure: TRUE or FALSE.
Preserve inherent parent / child relationships
  --      #orientation:
  --            #position: <vector>  pre-positioning of the
model
  --            #rotation: <vector>  pre-rotation of the
model
  --            #scale: <vector>  pre-scale of the model
  --      #highlight:  SETTINGS FOR MODEL HIGHLIGHT
  --            #cycle_duration: <milliseconds>.  Determines
the cycle time for this event
  --            #diffuse: <rgbvalue>.  Maximum diffuse color
of highlight shader
```

```
  --            #ambient: <rgbvalue>.  Maximum ambient color
of highlight shader
  -- #groups:  Pre-defined groups by name & members (models
& lights)


  tempModelOrientation = [#position: vector(0,0,0),
#rotation: vector(-90,0,0), #scale: vector(0.5,0.5,0.5)]

  tempWorldSettings = [#structure: [#preserve_structure:
FALSE], #highlight:[#cycle_duration:3000, #diffuse:
rgb(255,0,255), #ambient: rgb(0,0,50)], #orientation:
tempModelOrientation]

  tempModelSettings = [#worldsettings: tempWorldSettings]


  -- #LIGHTING SETTINGS holds the light settings for the
current model
  -- #worldsettings:
  --      #background_color: <hexvalue>
  --      #ambient_color: <hexvalue>
  --      #directional: ??? INCLUDE THIS ???
  -- tempLightingSettings =
[#worldsettings:[#background_color: "#000055",
#ambient_color: "#669933"]]
  -- ST. DENIS LIGHTING
  tempLightingSettings = [#worldsettings:[#background_color:
"#000000", #ambient_color:"#000000"]]

  -- #CAMERA SETTINGS holds the settings for the current
scene's camera
  -- #initial_position: holds the translation for the
initial orientation of the camera
  --       #x_pos: <float>  The x position of the camera in
units
  --       #y_pos: <float>  The y position of the camera in
units
  --       #z_pos: <float>  The z position of the camera in
units
  --       #pan_deg: <float>  The degrees to pan the camera
  --       #tilt_deg: <float>  The degrees to tilt the
camera
  -- #headlight:  holds the properties for the camera
headlight object
  --       #enabled: TRUE or FALSE.  Is the headlight
enabled?
  --       #type: #spot or #directional.  Type of light
being used.
  --       #color: <hexval>.  Color / Intensity of the
headlight.
  -- #fading:  holds the properties for the camera fades
  --       #default_color: <hexval>.  Default fade color
value
```

```
  --       #custom: <rgbcolor>.  List of members for "custom
fades"
  -- STANDARD SETTINGS
  -- tempInitialCameraPosition = [#x_pos: 1013, #y_pos:
185.7, #z_pos: -1137, #pan_deg: 140, #tilt_deg: 31.5]
  -- ST DENIS SETTINGS
  tempInitialCameraPosition = [#x_pos: 891.3, #y_pos: 331.0,
#z_pos: 335.2, #pan_deg:-90, #tilt_deg: 75]

  tempCameraFading = [#default_color: "#000000", #custom:[]]

  tempCameraSettings = [#initial_position:
tempInitialCameraPosition, #headlight: [#enabled: FALSE,
#type:#spot, #color:rgb(200,200,200)],
#fading:tempCameraFading]

  currentscene.settings = [#timeline: tempTimelineSettings,
#navigation: tempNavigationSettings, #model:
tempModelSettings, #lighting: tempLightingSettings, #camera:
tempCameraSettings]


  -- TEMPORARY NAVIGATIONAL SETTINGS
  user_keymap = [#move_forward: 30, #move_backward:31,
#move_left:28, #move_right:29, #rotate_up:119,
#rotate_down:115, #rotate_left:97, #rotate_right:100,
#mouse_look:109]
  user_speedsettings = [#move: 10, #rotate:1,
#mouse_speed:5, #mouse_invert:FALSE]




end




on kajoTemporarySettings

  -- TEMPORARY HOUSING FOR THE KAJO DEMONSTRATION SETTINGS

  sceneLayoutProps.x = 319
  sceneLayoutProps.y = 232
  sceneLayoutProps.w = 449
  sceneLayoutProps.h = 336

  scenemodels = [[#name: "gothic", #path: "C:\Documents and
Settings\tcorbett\Desktop\navetest3.W3D"],[#name:"kajo",
#path: the moviePath & "/proxmty.W3D"]]

  -- currentscene = [#scenenum: 1, #scenename: "gothic",
#settings:[#starttime: 0, #type: #internal,
#navigation:[#enabled:TRUE, #type:#free],
#mouse:[#enabled:TRUE, #buffer_percent:10], #position:FALSE,
#attributes:[:]]]

  -- set up the current scene (temporary population until
parameters are
  -- imported from the document or loaded by default

  currentscene = [#scenenum: 2, #scenename: "kajo",
#settings:[:]]



  -- #TIMELINE SETTINGS hold the settings for the timeline
  -- #starttime: starting milliseconds value (greater than
zero indicates returning to a scene)
  -- #type: #internal or #external (external denotes
timeline is A/V dependent)
  -- #attributes:  Holds the attributes for the A/V
dependant file (#external timeline only)
  --      #filepath: <file path>. AV File Location
(streaming / local)
  --      #start_time: <time>.   What location within the
file to start from
  --      #duration:  <time>.  Duration of this clip, or
#all (plays the entire duration)

  -- SAMPLE TIMELINE SETTINGS - NON-AV DEPENDENT
  tempTimelineSettings = [#starttime: 0, #type: #internal,
#attributes:[]]

  -- SAMPLE TIMELINE SETTINGS - AV DEPENDENT
  --  tempTimelineSettings = [#starttime: 0, #type:
#external, #attributes:[#filepath: #hold, #start_time: 0,
#duration: #all]]

  -- #NAVIGATION SETTINGS hold the nav settings for the
scene
  -- #enabled: TRUE or FALSE.  Is user-navigation enabled.
  -- #type: #free, ?, or ?.  User-navigation style (free
denotes flythrough)
  -- #mouse: Mouse Navigational Settings
  --      #enabled: TRUE or FALSE.  Is the mouse-look
feature enabled
  --      #buffer_percent: <num>.  Percentage-of-sprite area
(centered) where mouse-look will not occur.
  -- #restrictions:
  --      #z-axis: TRUE or FALSE.  Can the user fly-up and
down in free-form?
  -- #boundary: Model Boundary Settings
  --      #enabled: TRUE or FALSE.  Is the boundary
condition setting on?
  -- #collision: Model Collision Settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the collision setting
on?
  -- #position: Position determination settings / Parameters
  --      #enabled: TRUE or FALSE.  Is the position-event
feature on?

  -- SAMPLE TEMP NAV SETTINGS - FREE EXPLORE
  tempNavigationSettings = [#enabled: TRUE, #type: #free,
#restrictions:[#zaxis: TRUE], #mouse:[#enabled: TRUE,
#buffer_percent:10], #boundary:[#enabled:FALSE],
#collision:[#enabled:FALSE], #position:[#enabled:FALSE]]

  -- SAMPLE TEMP NAV SETTINGS - SLAVE EXPLORE
  --  tempNavigationSettings = [#enabled: FALSE, #type:
#free, #restrictions:[#zaxis: FALSE], #mouse:[#enabled:
TRUE, #buffer_percent:10], #boundary:[#enabled:FALSE],
#collision:[#enabled:FALSE], #position:[#enabled:FALSE]]


  -- #MODEL SETTINGS holds the settings for the current
model
  -- #path: <dirpath>.  ???? SHOULD I EVEN DO THIS ?????
  -- #worldsettings:
  --      #structure:
  --            #preserve_structure: TRUE or FALSE.
Preserve inherent parent / child relationships
  --      #orientation:
  --            #position: <vector>  pre-positioning of the
model
  --            #rotation: <vector>  pre-rotation of the
model
  --            #scale: <vector>  pre-scale of the model
  --      #highlight:  SETTINGS FOR MODEL HIGHLIGHT
  --            #cycle_duration: <milliseconds>.  Determines
the cycle time for this event
  --            #diffuse: <rgbvalue>.  Maximum diffuse color
of highlight shader
  --            #ambient: <rgbvalue>.  Maximum ambient color
of highlight shader
  -- #groups:  Pre-defined groups by name & members (models
& lights)


  -- tempModelOrientation = [#position: vector(0,0,0),
#rotation: vector(-90,0,0), #scale: vector(0.5,0.5,0.5)]

  tempModelOrientation = [#position: vector(0,0,0),
#rotation: vector(-90,0,0), #scale: vector(1,1,1)]


  tempWorldSettings = [#structure: [#preserve_structure:
FALSE], #highlight:[#cycle_duration:3000, #diffuse:
rgb(255,0,255), #ambient: rgb(0,0,50)], #orientation:
tempModelOrientation]
```

```
  tempModelSettings = [#worldsettings: tempWorldSettings]


  -- #LIGHTING SETTINGS holds the light settings for the
current model
  -- #worldsettings:
  --      #background_color: <hexvalue>
  --      #ambient_color: <hexvalue>
  --      #directional: ??? INCLUDE THIS ???
  tempLightingSettings = [#worldsettings:[#background_color:
"#000000", #ambient_color: "#000000"]]


  -- #CAMERA SETTINGS holds the settings for the current
scene's camera
  -- #initial_position: holds the translation for the
initial orientation of the camera
  --       #x_pos: <float>  The x position of the camera in
units
  --       #y_pos: <float>  The y position of the camera in
units
  --       #z_pos: <float>  The z position of the camera in
units
  --       #pan_deg: <float>  The degrees to pan the camera
  --       #tilt_deg: <float>  The degrees to tilt the
camera
  --       #fieldofview: <float> The camera field of view in
degrees
  -- #headlight:  holds the properties for the camera
headlight object
  --       #enabled: TRUE or FALSE.  Is the headlight
enabled?
  --       #type: #spot or #directional.  Type of light
being used.
  --       #color: <hexval>.  Color / Intensity of the
headlight.
  -- #fading:  holds the properties for the camera fades
  --       #default_color: <hexval>.  Default fade color
value
  --       #custom: <rgbcolor>.  List of members for "custom
fades"
  tempInitialCameraPosition = [#x_pos: 927.63, #y_pos:
59.97, #z_pos: 132.33, #pan_deg: 52.33, #tilt_deg: 0,
#fieldofview: 47.07]
  tempCameraFading = [#default_color: "#000000", #custom:[]]

  tempCameraSettings = [#initial_position:
tempInitialCameraPosition, #headlight: [#enabled: FALSE,
#type:#spot, #color:rgb(200,200,200)],
#fading:tempCameraFading]

  currentscene.settings = [#timeline: tempTimelineSettings,
#navigation: tempNavigationSettings, #model:
```

```
tempModelSettings, #lighting: tempLightingSettings, #camera:
tempCameraSettings]


  -- TEMPORARY NAVIGATIONAL SETTINGS
  user_keymap = [#move_forward: 30, #move_backward:31,
#move_left:28, #move_right:29, #rotate_up:119,
#rotate_down:115, #rotate_left:97, #rotate_right:100,
#mouse_look:109]
  user_speedsettings = [#move: 5, #rotate:4, #mouse_speed:5,
#mouse_invert:FALSE]




end
```

Shockwave3D Status

```
on check3Dready whichmember
  if whichmember.state = 4 then  -- a state of 4 tells you
the media is ready
    return true                -- if it is ready, return
true (for error checking & debugging)
  else
    return false               --if it isn't ready return
false (for error checking & debugging)
  end if
end

on clearscene whichmember          -- note: takes one
argument in the form of     member("which3Dcastmember")
  if check3Dready(whichmember) then    -- just some last
minute double checking to make sure the SW3D is ready
    whichmember.resetworld()       -- if everything is
ready, reset the world to its default values
    return true                  -- return true (for error
checking & debugging)
  else
    return false                 -- the SW3D was not ready,
so return false (for error checking & debugging)
  end if
end
```

3D INIT SCRIPTS

```
global scene

on exitFrame me
  put scene.state

  if check3Dready(scene) then     -- call custom handler:  is
the SW3D castmember ready?


    -- on to the run-screen
    go "3dReady"
  else
    go the frame                  -- if it is not ready, wait
here
  end if
end
```

Model / Group Interaction Scripts

```
global scene
global current_model_list
global current_group_list
global current_light_list
global invisible_model_list
global off_light_list
global scenemodels
global custom_shader_list
global model_orientation_list
global group_orientation_list


on catalogModelNames
  -- populates the CURRENT MODEL LIST with name of each
model within world.
  current_model_list = []
  repeat with i = 1 to scene.model.count
    current_model_list.append(scene.model[i].name)
  end repeat
end

on verifyModel model_name
  -- verifies the existance of a model within the world
  if (current_model_list.getOne(model_name)) then
    return TRUE
  else
    return FALSE
  end if
end

on catalogGroupNames
  -- populates the CURRENT GROUP LIST with name of each
model within world
  current_group_list = []
  repeat with i = 1 to scene.group.count
    current_group_list.append(scene.group[i].name)
  end repeat
end

on verifyGroup group_name
  -- verifies the existance of a group within the world
  if (current_group_list.getOne(group_name)) then
    return TRUE
  else
    return FALSE
  end if
end


on getExternalModelPath scenenumber
  set xstring = ""
  repeat with a = 1 to scenemodels.count
    xstring = xstring & scenemodels[a].name &&
scenemodels[a].filepath

  end repeat


  set mod_loc = scenemodels[scenenumber].filepath
  sendStatus(xstring)
  -- sendStatus("External Model Location: " & mod_loc)
  return mod_loc
end

on getModelPosition model_name
  -- verifies that the model exists, and if so, returns the
position as a vector
  if (verifyModel(model_name)) then
    set this_vec =
scene.model(model_name).transform.position
    return this_vec
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on getModelRotation model_name
  -- verifies that the model exists, and if so, returns the
rotation.
  if (verifyModel(model_name)) then
    set this_vec =
scene.model(model_name).transform.rotation
    return this_vec
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on getModelDirection model_name
  -- verifies that the model exists, and if so, returns the
direction
  if (verifyModel(model_name)) then
    set this_direction =
scene.model(model_name).pointAtOrientation[1]
    set this_axisangle =
scene.model(model_name).getWorldTransform().axisAngle

    set world_direction =
rotateDirectionAboutAxis(this_direction, this_axisangle[1],
this_axisangle[2])

    return world_direction

  end if
end
```

```
on getModelScale model_name
  -- verifies that the model exists, and if so, returns the
scale.
  if (verifyModel(model_name)) then
    set this_vec = scene.model(model_name).transform.scale
    return this_vec
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on setModelPosition model_name, model_position
  -- verifies that model exists, if so moves it to the new
position vector
  if (verifyModel(model_name)) then
    set this_trans =
scene.model(model_name).getWorldTransform().position
    set translate_vec = model_position - this_trans
    scene.model(model_name).translate(translate_vec, #world)
  else
    return -1  -- denotes the model doesn't exist
  end if
end

on setModelSelfRotation model_name, rotation_axis,
rotation_angle
  -- verifies that the model exists, if so rotates it about
the self-axis
  if (verifyModel(model_name)) then
    scene.model(model_name).rotate(vector(0,0,0),
rotation_axis, rotation_angle, #self)
  else
    return -1  -- denotes the model doesn't exist
  end if
end

on setModelWorldRotation model_name, rotation_axis,
rotation_angle
  -- verifies that the model exists, and if so, rotates it
about the world defined axis
  -- don't need to check the axis for a 0 value, because
director ignores this command
  if (verifyModel(model_name)) then
    set posVec =
scene.model(model_name).getWorldTransform().position
    scene.model(model_name).rotate(posVec, rotation_axis,
rotation_angle, #world)
  else
    return -1  -- denotes the model doesn't exist
  end if

end
```

```
on pointModelAtPosition model_name, target_position
  -- verifies that model exists, if so changes the rotation
to match the new vector
  if (verifyModel(model_name)) then
    -- check for an invalid position (identical to current
point)
    if (target_position <>
scene.model(model_name).getWorldTransform().position) then
      scene.model(model_name).pointAt(target_position)
    end if
  else
    return -1 -- denotes the model doesn't exist
  end if
end

on pointModelAtDirection model_name, target_direction
  -- verifies that model exists, if so changes the rotation
to match the new vector
  if (verifyModel(model_name)) then
    -- check for an invalid position (identical to current
point)
    set model_position =
scene.model(model_name).getWorldTransform().position
    set target_position = model_position + target_direction
    if vectorP(target_direction) then
      if (target_direction.magnitude <> 0) then
        scene.model(model_name).pointAt(target_position)
      end if
    end if


  else
    return -1 -- denotes the model doesn't exist
  end if

end

on rotateModelFromDirection model_name, axis_vector, angle
  -- verifies that the model exists, if so rotates it, but
counter rotates the point at direction
  if (verifyModel(model_name)) then
    -- rotate the model
    scene.model(model_name).rotate(vector(0,0,0),
axis_vector, angle, #self)
    -- counter-rotate the direction
    rotateModelOrientation(model_name, axis_vector, (-1 *
angle))
  else
    return -1  -- denotes the model doesn't exist
  end if

end
```

```
on rotateModelOrientation model_name, axis_vector, angle
  -- verifies the model exists, then rotates around the axis
vector by angle (degrees)
  if (verifyModel(model_name)) then
    -- is this in the model_orientation_list?
    set this_orientation =
model_orientation_list.getaProp(model_name)

    set dir_vec =
scene.model(model_name).pointAtOrientation[1]
    set up_vec =
scene.model(model_name).pointAtOrientation[2]

    if voidP(this_orientation) then
      -- set the orientation hold values
      model_orientation_list.addProp(model_name, [dir_vec,
up_vec])
    end if

    -- set up the transforms
    set dir_trans = transform()
    set up_trans = transform()

    dir_trans.position = dir_vec
    up_trans.position = up_vec

    -- rotate about the given axis
    dir_trans.rotate(vector(0,0,0), axis_vector, angle)
    up_trans.rotate(vector(0,0,0), axis_vector, angle)

    -- apply the new values
    --    scene.model(model_name).pointAtOrientation =
[getnormalized(dir_trans.rotation),
getnormalized(up_trans.rotation)]
    scene.model(model_name).pointAtOrientation =
[dir_trans.position, up_trans.position]

  else
    return -1  -- denotes the model doesn't exist
  end if
end

on resetModelOrientation model_name
  -- verifies that the model exists, then resets to the
original values
  if (verifyModel(model_name)) then
    set this_orientation =
model_orientation_list.getaProp(model_name)
    if NOT(voidP(this_orientation)) then
      scene.model(model_name).pointAtOrientation =
this_orientation
      model_orientation_list.deleteProp(model_name)
    end if
  else
    return -1
  end if

end


on setModelScale model_name, model_scale
  -- verifies that model exists, if so changes the scale to
match the new vector
end

on getGroupPosition group_name
  -- verifies that the group exists, and if so, returns the
position.

  if (verifyGroup(group_name)) then
    set this_vec =
scene.group(group_name).transform.position
    return this_vec
  else
    return -1   -- denotes the group doesn't exist
  end if
end

on getGroupRotation group_name
  -- verifies that the group exists, and if so, returns the
rotation.
  if (verifyGroup(group_name)) then
    set this_vec =
scene.group(group_name).transform.rotation
    return this_vec
  else
    return -1   -- denotes the group doesn't exist
  end if
end

on getGroupScale group_name
  -- verifies that the group exists, and if so, returns the
scale.
  if (verifyGroup(group_name)) then
    set this_vec = scene.group(group_name).transform.scale
    return this_vec
  else
    return -1   -- denotes the group doesn't exist
  end if
end

on getGroupDirection group_name
  -- verifies that the group exists, and if so, returns the
direction
  if (verifyGroup(group_name)) then
```

```
    set this_direction =
scene.group(group_name).pointAtOrientation[1]
    set this_axisangle =
scene.group(group_name).getWorldTransform().axisAngle

    set world_direction =
rotateDirectionAboutAxis(this_direction, this_axisangle[1],
this_axisangle[2])

    return world_direction

  end if
end

on setGroupPosition group_name, group_position
  -- verifies that group exists, if so moves it to the new
position vector
  if (verifyGroup(group_name)) then
    set this_trans =
scene.group(group_name).getWorldTransform().position
    set translate_vec = group_position - this_trans
    scene.group(group_name).translate(translate_vec, #world)
  else
    return -1  -- denotes the group doesn't exist
  end if

end

on setGroupSelfRotation group_name, rotation_axis,
rotation_angle
  -- verifies that the group exists, if so rotates it about
the self-axis
  if (verifyGroup(group_name)) then
    scene.group(group_name).rotate(vector(0,0,0),
rotation_axis, rotation_angle, #self)
  else
    return -1  -- denotes the group doesn't exist
  end if
end

on setGroupWorldRotation group_name, rotation_axis,
rotation_angle
  -- verifies that the group exists, and if so, rotates it
about the world defined axis
  -- don't need to check the axis for a 0 value, because
director ignores this command
  if (verifyGroup(group_name)) then
    set posVec =
scene.group(group_name).getWorldTransform().position
    scene.group(group_name).rotate(posVec, rotation_axis,
rotation_angle, #world)
  else
    return -1  -- denotes the group doesn't exist
  end if
```

```
end

on pointGroupAtPosition group_name, target_position
  -- verifies that group exists, if so changes the rotation
to match the new vector
  if (verifyGroup(group_name)) then
    -- check for an invalid position (identical to current
point)
    if (target_position <>
scene.group(group_name).getWorldTransform().position) then
      scene.group(group_name).pointAt(target_position)
    end if
  else
    return -1 -- denotes the group doesn't exist
  end if
end

on pointGroupAtDirection group_name, target_direction
  -- verifies that group exists, if so changes the rotation
to match the new vector
  if (verifyGroup(group_name)) then
    -- check for an invalid position (identical to current
point)
    set group_position =
scene.group(group_name).getWorldTransform().position
    set target_position = group_position + target_direction
    if vectorP(target_direction) then
      if (target_direction.magnitude <> 0) then
        scene.group(group_name).pointAt(target_position)
      end if
    end if

  else
    return -1 -- denotes the group doesn't exist
  end if

end

on rotateGroupFromDirection group_name, axis_vector, angle
  -- verifies that the group exists, if so rotates it, but
counter rotates the point at direction
  if (verifyGroup(group_name)) then
    -- rotate the group
    scene.group(group_name).rotate(vector(0,0,0),
axis_vector, angle, #self)
    -- counter-rotate the direction
    rotateGroupOrientation(group_name, axis_vector, (-1 *
angle))
  else
    return -1  -- denotes the group doesn't exist
  end if

end
```

```
on rotateGroupOrientation group_name, axis_vector, angle
  -- verifies the group exists, then rotates around the axis
vector by angle (degrees)
  if (verifyGroup(group_name)) then
    -- is this in the group_orientation_list?
    set this_orientation =
group_orientation_list.getaProp(group_name)

    set dir_vec =
scene.group(group_name).pointAtOrientation[1]
    set up_vec =
scene.group(group_name).pointAtOrientation[2]

    if voidP(this_orientation) then
      -- set the orientation hold values
      group_orientation_list.addProp(group_name, [dir_vec,
up_vec])
    end if

    -- set up the transforms
    set dir_trans = transform()
    set up_trans = transform()

    dir_trans.position = dir_vec
    up_trans.position = up_vec

    -- rotate about the given axis
    dir_trans.rotate(vector(0,0,0), axis_vector, angle)
    up_trans.rotate(vector(0,0,0), axis_vector, angle)

    -- apply the new values
    scene.group(group_name).pointAtOrientation =
[dir_trans.position, up_trans.position]

  else
    return -1  -- denotes the group doesn't exist
  end if
end

on resetGroupOrientation group_name
  -- verifies that the group exists, then resets to the
original values
  if (verifyGroup(group_name)) then
    set this_orientation =
group_orientation_list.getaProp(group_name)
    if NOT(voidP(this_orientation)) then
      scene.group(group_name).pointAtOrientation =
this_orientation
      group_orientation_list.deleteProp(group_name)
    end if

  else
```

```
    return -1
  end if

end


on getModelVisibility model_name
  -- verifies that the model exists, and if so, returns the
current visibility
  if (verifyModel(model_name)) then
    return scene.model(model_name).visibility
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on DisIncludeModel model_name
  -- verifies that the model exists, and if so, turns it off
and removes it from
  -- the current_model_list
  if (verifyModel(model_name)) then
    scene.model(model_name).visibility = #none
    current_model_list.deleteOne(model_name)
    return TRUE
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on DisIncludeGroup group_name
  -- verifies that the model exists, and if so, turns it off
and removes it from
  -- the current_model_list
  if (verifyGroup(group_name)) then
    scene.deletegroup(group_name)
    current_group_list.deleteOne(group_name)
    return TRUE
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on DisIncludeLight light_name
  -- verifies that the model exists, and if so, turns it off
and removes it from
  -- the current_model_list
  if (verifyLight(light_name)) then
    scene.deleteLight(light_name)
    current_light_list.deleteOne(light_name)
    return TRUE
  else
    return -1   -- denotes the model doesn't exist
  end if
end
```

```
on TurnOffModel model_name
  -- verifies that the model currently exists, and that it
is currently visible
  -- then turns it off, and catalogs it.
  if (verifyModel(model_name)) then
    if voidP(invisible_model_list.getaProp(model_name)) and
(scene.model(model_name).visibility <> #none) then
      invisible_model_list.addProp(model_name,
scene.model(model_name).visibility)
      scene.model(model_name).visibility = #none

      return TRUE

    else if
NOT(voidP(custom_shader_list.getaProp(model_name))) then
      -- model is off, custom shader is on.
      model_clone = custom_shader_list.getaProp(model_name)
      scene.model(model_clone.clone_name).visibility = #none
      invisible_model_list.addProp(model_name,
model_clone.visibility)

      return TRUE

    else
      return FALSE  -- denotes that the model is already off
    end if
  else
    return -1   -- denotes the model doesn't exist
  end if

end

on TurnOnModel model_name
  -- verifies that the model exists, and is currently
invisible, and if so
  -- turns it back on
  if (verifyModel(model_name)) then
    set this_visibility =
invisible_model_list.getaProp(model_name)
    if NOT(voidP(this_visibility)) then

      -- this is on the list
      invisible_model_list.deleteProp(model_name)

      set model_shader =
custom_shader_list.getaProp(model_name)
      if (voidp(model_shader)) then
        -- if this model doesn't have a custom shader, turn
it on
        scene.model(model_name).visibility = this_visibility
      else
        -- otherwise, turn on the custom shader instead
        scene.model(model_shader.clone_name).visibility =
this_visibility
      end if

      return TRUE

    else
      return FALSE   -- model is not currently in the
invisible list
    end if
  else
    return -1   -- denotes the model doesn't exist
  end if

end

on createNewGroup group_name
  -- verify that group name does not currently exist, and
creates it
  if (verifyGroup(group_name)) then
    return FALSE   -- denotes group already exists
  else
    scene.newGroup(group_name)
    current_group_list.append(group_name)
    return TRUE
  end if
end

on addModelToGroup group_name, model_name
  -- verify that group and model exist, if so, add model to
group
  if (group_name <> "World") then

    if (verifyGroup(group_name)) then
      if (verifyModel(model_name)) then

        -- change the parent of the model

scene.group(group_name).addChild(scene.model(model_name),
#preserveWorld)

        return TRUE
      else
        return 0  -- denotes that model doesn't exist
      end if
    else
      return -1  -- denotes that group doesn't exist
    end if
  else
    return -1  -- denotes invalid function
  end if
end

on addLightToGroup group_name, light_name
```

```
  -- verify that the group and light exist, if so, add light
to group
  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      if (verifyLight(light_name)) then

        -- change the parent of the light

scene.group(group_name).addChild(scene.light(light_name),
#preserveWorld)

        return TRUE
      else
        return 0 -- denotes that light does not exist
      end if
    else
      return -1  -- denotes that group doesn't exist
    end if
  else
    return -1   -- denotes invalid function
  end if
end



on deleteModelFromGroup group_name, model_name
  -- verify that group and model exist, and that model is in
group
  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      -- catalog this group
      set in_group = []
      set imax = scene.group(group_name).child.count
      repeat with i = 1 to iMax

in_group.append(scene.group(group_name).child[i].name)
      end repeat

      -- check the catalog to see if this is in it
      if (in_group.getOne(model_name)) then

scene.group("World").addChild(scene.model(model_name),
#preserveWorld)
        return TRUE
      else
        return 0   -- denotes model isn't in group
      end if
    else
      return -1   -- denotes group doesn't exist
    end if
  else
    return -1   -- denotes invalid function
  end if
```

```
end

on deleteLightFromGroup group_name, light_name
  -- verify that group and light exist, and that light is in
group
  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      -- catalog this group
      set in_group = []
      set imax = scene.group(group_name).child.count
      repeat with i = 1 to iMax

in_group.append(scene.group(group_name).child[i].name)
      end repeat

      -- check the catalog to see if this is in it
      if (in_group.getOne(light_name)) then

scene.group("World").addChild(scene.light(light_name),
#preserveWorld)
        return TRUE
      else
        return 0   -- denotes light isn't in group
      end if
    else
      return -1  -- denotes group doesn't exist
    end if
  else
    return -1   -- denotes invalid function
  end if

end


on destroyGroup group_name
  -- verifies that the group currently exists, then changes
the parent
  -- to the group("World")
  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      -- set model parents to "World" group
      set i = scene.group(group_name).child.count
      repeat while i > 0

scene.group("World").addChild(scene.group(group_name).child[
i], #preserveWorld)
        i = i - 1
      end repeat
      scene.deleteGroup(group_name)
      catalogGroupNames()
      return TRUE
    else
      return -1   -- denotes group doesn't exist
    end if
  else
    return -1   -- denotes invalid function
  end if
end


on turnOffGroup group_name
  -- verifies that the group exists, then turns off the
visibility of
  -- each model within the group

  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      set iMax = scene.group(group_name).child.count
      repeat with i = 1 to iMax
        set this_child =
scene.group(group_name).child[i].name
        if (verifyModel(this_child)) then
          -- this is a model, turn off the visibility
          turnOffModel(this_child)
        end if
      end repeat
      return TRUE
    else
      return -1   -- denotes group doesn't exist
    end if
  else
    return -1   -- this denotes an invalid function
  end if
end

on turnOnGroup group_name
  -- verifies that the group exists, then turns on the
visibility of each
  -- invisible model within the group

  if (group_name <> "World") then
    if (verifyGroup(group_name)) then
      set iMax = scene.group(group_name).child.count
      repeat with i = 1 to iMax
        set this_child =
scene.group(group_name).child[i].name
        if (verifyModel(this_child)) then
          -- this is a model, turn on the visibility
          turnOnModel(this_child)
        end if
      end repeat
      return TRUE
    else
      return -1   -- denotes group doesn't exist
    end if
  else
    return -1   -- denotes an invalid function
  end if
```

```
end


on resetInherentStructure
  -- removes all parent / child relationships by setting all
parents to "World"
  -- preserves world transforms

  -- destroy all groups
  set g_count = scene.group.count
  repeat while g_count > 0
    destroyGroup(scene.group[g_count].name)
    g_count = g_count - 1
  end repeat

  -- reset all models to world parent
  set m_max = scene.model.count
  repeat with m_count = 1 to m_max
    scene.group("World").addChild(scene.model[m_count],
#preserveWorld)
  end repeat

  -- reset all lights to world parent
  set l_max = scene.light.count
  repeat with l_count = 1 to l_max
    scene.group("World").addChild(scene.model[l_count],
#preserveWorld)
  end repeat

end

on createGroupClone group_name, clone_name
  -- verifies that this group exists,
  -- sets parent of all group's children to new group
  -- updates the current_group_list
  -- returns the group itself
  if (verifyGroup(group_name)) then
    -- create the new group
    group_created = createNewGroup(clone_name)
    if (group_created) then
      set c_count = scene.group(group_name).child.count
      repeat while c_count > 0
        -- add the child to the clone group

scene.group(clone_name).addChild(scene.group(group_name).chi
ld[c_count], #preserveWorld)
        c_count = c_count - 1
      end repeat
      return TRUE
    else
      return 0   -- denotes that this group name already
exists
    end if
  else
```

```
    return -1  -- denotes that this group doesn't exist
  end if

end

on runtimePositionSet event_attributes
  set obj_name = event_attributes.name
  set obj_type = event_attributes.type
  set obj_position = event_attributes.position

  case (obj_type) of:
    #camera: setCameraPosition(obj_position)
    #model: setModelPosition(obj_name, obj_position)
    #group: setGroupPosition(obj_name, obj_position)
    #light: setLightPosition(obj_name, obj_position)
  end case

  return TRUE

end
```

Event Queue System

```
global eventQueue
global timeline


on resetEventQueue
  -- initialized the queued event
  eventQueue = []
end

on doQueuedEvents
  -- runs through a list of queued events
  -- passes each event, event properties, and the current
time through the processEvent command
  -- looks for a Boolean return from the function
  -- if the function returns true, then the event is removed
from the queue
  -- otherwise, the event remains in the queue until it is
successfully executed


  if listP(eventQueue) then
    if eventQueue.count > 0 then

      -- evaluate the current times
      set current_time = timeline.current

      -- run through the queue
      qcount = 1
      repeat while (qcount <= eventQueue.count)

        set queuedEvent = eventQueue[qcount]

        -- check for HOLD events
        if (queuedEvent.starttime = #hold) then
          -- if the name not found, change the launch time
to now.
          set check_name = queuedEvent.hold_trigger
          set name_found = FALSE

          repeat with name_count = 1 to eventQueue.count
            if (eventQueue[name_count].name = check_name)
then name_found = TRUE
          end repeat

          if (name_found) then
            -- trigger event still in queue, jump to the
next event
            qcount = qcount + 1
          else
            -- ready to go, will launch at next repeat
            queuedEvent.starttime = #now
          end if

        else if (queuedEvent.starttime = #now) then
          -- check for NOW events
          queuedEvent.starttime = current_time

        else if (queuedEvent.starttime > current_time) then
-- check to see if event should start yet

          qcount = qcount + 1

        else   -- not on hold



          -- check to see if this event is complete
          if (queuedEvent.duration <> #expired) then
            if ((queuedEvent.duration = 0) or (current_time
>= (queuedEvent.starttime + queuedEvent.duration))) then
              queuedEvent.duration = #expired
            end if
          end if

          -- set the percentage
          if (queuedEvent.duration = #expired) then
            set percentage = 1
          else
            set percentage = ((float(current_time) -
queuedEvent.starttime) / queuedEvent.duration)
          end if

          -- reset the verification response
          set qsuccess = #empty

          -- send the event to be evaluated & processed
          qsuccess = processEvent(qcount,
queuedEvent.command, queuedEvent.attributes, percentage)

          set qremove = FALSE

          -- check the verification response
          if (qsuccess = #cancel) then  -- has it been
cancelled?
            qremove = TRUE  -- remove it from the queue
          else if (qsuccess = #true) then
            if (queuedEvent.priority = #once) then  -- did
this execute?
              qremove = TRUE  -- remove it from the queue
            else if ((queuedEvent.priority = #verify) or
(queuedEvent.priority = #noverify)) then  -- has this been
verified
              if (queuedEvent.duration = #expired) then  --
has the duration passed?
                qremove = TRUE  -- remove it from the queue
              end if
            end if
          else   -- qsuccess returned #false
            if (queuedEvent.priority = #noverify) then  --
verification doesn't matter
              if (queuedEvent.duration = #expired) then --
has the duration passed?
                qremove = TRUE  -- remove it from the queue
              end if
            end if
          end if

          if (qremove) then
            -- remove this item from the queue
            eventQueue.deleteAt(qcount)
          else
            -- update the queued item and the count
            eventQueue[qcount] = queuedEvent
            qcount = qcount + 1
          end if

        end if

      end repeat

    end if
  end if


end


on addEventToQueue event
  -- adds the event to the queue

  -- events in the queue have the following structure:
  -- [#name, #command, #starttime, #duration, #priority,
#attributes]
  -- #name:  unique name for this event (#hold priority
items check for this completion)
  -- #command:  event function name / type, ex: #fade_camera
  -- #starttime: <time>.  The milliseconds start time for
the event
  --              OR  #now.  Starts at the current time, when
added to queue
  --              OR  #hold.  Hold this until the item in the
"hold_trigger" is removed from the queue
  --              OR  #prescene.  Executes prior to launching
the scene.
  -- #duration: <time>.  The milliseconds that this will
operate
  -- #priority: #ongoing.  This event repeats until
cancelled.
```

```
  --            OR #verify.  This event repeats until
complete & verified
  --            OR #noverify.  This event repeats until
duration is passed.
  --            OR #once.  This event repeats until first
verification.
  -- #attributes:  <event specific attributes>
  -- #hold_trigger: <other event name>


  eventQueue.append(event)

end

on processEvent event_number, event_name, event_attributes,
percent_complete

  -- set the verification object
  set event_response = #false

  case (event_name) of
    #fade_camera:
      set fade_complete = cameraFade(event_attributes,
percent_complete)
      if (fade_complete) then event_response = #true

    #change_lightcolor:
      set change_response =
changeLightColor(event_attributes, percent_complete)
      if (change_response = TRUE) then
        event_response = #true
      else if (change_response = -1) then
        event_response = #cancel
      else if (listP(change_response)) then
        eventQueue[event_number].attributes =
change_response
      end if

    #toggle_highlight:
      if (event_attributes.toggle = #on) then
        highlightObject(event_attributes.object_name)
      else
        unHighlightObject(event_attributes.object_name)
      end if

      set event_response = #true

    #highlight_object:
      highlightObject(event_attributes.object_name)
      set event_response = #true

    #reset_highlight:
      unHighlightObject(event_attributes.object_name)
      set event_response = #true
    #toggle_light:
      if (event_attributes.toggle = #on) then
        turnOnLight(event_attributes.light_name)
      else if (event_attributes.toggle = #off) then
        turnOffLight(event_attributes.light_name)
      end if

      set event_response = #true

    #change_opacity:
      set opacity_response =
changeModelOpacity(event_attributes, percent_complete)
      if (opacity_response = TRUE) then
        event_response = #true
      else if (opacity_response = -1) then
        event_response = #cancel
      else if (listP(opacity_response)) then
        eventQueue[event_number].attributes =
opacity_response
      end if

    #change_groupopacity:
      set opacity_response =
changeGroupOpacity(event_attributes, percent_complete)
      if (opacity_response = TRUE) then
        event_response = #true
      else if (opacity_response = -1) then
        event_response = #cancel
      else if (listP(opacity_response)) then
        eventQueue[event_number].attributes =
opacity_response
      end if



    #toggle_model:
      if (event_attributes.toggle = #on) then
        turnOnModel(event_attributes.model_name)
      else if (event_attributes.toggle = #off) then
        turnOffModel(event_attributes.model_name)
      end if

      set event_response = #true

    #toggle_group:
      if (event_attributes.toggle = #on) then
        turnOnGroup(event_attributes.group_name)
      else if (event_attributes.toggle = #off) then
        turnOffGroup(event_attributes.group_name)
      end if

      set event_response = #true
    #change_modelcolor:

      set color_response = 
changeModelColor(event_attributes, percent_complete)
      if (color_response = TRUE) then
        event_response = #true
      else if (color_response = -1) then
        event_response = #cancel
      else if (listP(color_response)) then
        eventQueue[event_number].attributes = color_response
      end if

    #change_groupcolor:

      set color_response = 
changeGroupColor(event_attributes, percent_complete)
      if (color_response = TRUE) then
        event_response = #true
      else if (color_response = -1) then
        event_response = #cancel
      else if (listP(color_response)) then
        eventQueue[event_number].attributes = color_response
      end if


    #linear_motion:
      set motion_response = 
interpolateLinearMotion(event_attributes, percent_complete)
      if (motion_response = TRUE) then
        event_response = #true
      else if (motion_response = -1) then
        event_response = #cancel
      else if (listP(motion_response)) then
        eventQueue[event_number].attributes = 
motion_response
      end if

    #orbit_motion:
      set motion_respone = 
interpolateOrbitMotion(event_attributes, percent_complete)
      if (motion_response = TRUE) then
        event_response = #true
      else if (motion_response = -1) then
        event_response = #cancel
      else if (listP(motion_response)) then
        eventQueue[event_number].attributes = 
motion_response
      end if

    #spline_motion:
      set motion_response = 
interpolateSplineMotion(event_attributes, percent_complete)
      if (motion_response = TRUE) then
        event_response = #true
      else if (motion_response = -1) then
        event_response = #cancel
      else if (listP(motion_response)) then
        eventQueue[event_number].attributes =
motion_response
      end if

    #rotate_object:
      set rotation_response =
interpolateObjectRotation(event_attributes,
percent_complete)
      if (rotation_response = TRUE) then
        event_response = #true
      else if (rotation_response = -1) then
        event_response = #cancel
      else if (listP(rotation_response)) then
        eventQueue[event_number].attributes =
rotation_response
      end if

    #rotate_camera:
      set rotation_response =
interpolateCameraRotation(event_attributes,
percent_complete)
      if (rotation_response = TRUE) then
        event_response = #true
      else if (rotation_response = -1) then
        event_response = #cancel
      else if (listP(rotation_response)) then
        eventQueue[event_number].attributes =
rotation_response
      end if

    #reset_orientation:
      set reset_response =
executeResetObjectOrientation(event_attributes)
      if (reset_response = TRUE) then
        event_response = #true
      else if (reset_response = -1) then
        event_response = #cancel
      end if


    #target_transition:
      set transition_response =
interpolateTargetTransition(event_attributes,
percent_complete)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      else if (listP(transition_response)) then
        eventQueue[event_number].attributes =
transition_response
```

```
      end if

    #call_motion:
      set transition_response =
callMotionFromCatalog(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      end if

    #set_position:
      set transition_response =
runtimePositionSet(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      end if

    #register_click_event:
      set transition_response =
registerClickTrigger(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      end if

    #unregister_click_event:
      set transition_response =
unregisterClickTrigger(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      end if

    #register_prox_event:
      set transition_response =
registerProxTrigger(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response =  -1) then
        event_response = #cancel
      end if

    #unregister_prox_event:
      set transition_response =
unregisterProxTrigger(event_attributes)
      if (transition_response = TRUE) then
        event_response = #true
      else if (transition_response = -1) then
        event_response = #cancel
      end if
  end case
  return event_response
end
```

Timeline System

```
-- THIS SYSTEM NEEDS TO BE RE-EDITED TO INTERACT WITH / CALL
FROM A BEHAVIOR SCRIPT
-- THAT WILL BE ATTACHED TO THE 3D MEMBER.



global timeline
global currentscene
global loopstarthold
global sceneLayoutProps
global model_highlight_list
global av_object_list
global av_object
global timehold
global prox_trigger_list



on resetTimeline

  -- TIMELINE global explanation
  -- timeline holds the major props to be tested & used in
the command loop
  -- #start holds the start-time for this scene
  --      (allows for mid-scene starts, i.e. returning to
scene from side presentation)
  -- #current holds the current time for the internal
timeline in milliseconds
  -- #type indicates timeline type (#internal, or #external)
  -- #navigation holds the boolean for testing keys / mouse
  -- #position holds the boolean for testing position in
space
  -- #attributes holds #external av_object name,

  -- sets the defaults for the TIMELINE global
  timeline = [#start:0, #current:0, #type:#internal,
#navigation: FALSE, #position:FALSE, #attributes: [:]]

end

on initializeTimeline
  sendStatus("Initializing timeline")
  -- start (or restart) the command loop by resetting the
timeline
  resetTimeline()

  -- get the attributes from the currentscene
  timeline.start = currentscene.settings.timeline.starttime
  timeline.type = currentscene.settings.timeline.type
  timeline.navigation =
currentscene.settings.navigation.enabled
```

```
  timeline.position =
currentscene.settings.navigation.position.enabled

  if (timeline.type = #external) then
    -- get the timeline information
    av_idx = getNamedListIndex(av_object_list,
currentscene.settings.timeline.attributes.name)
    -- initialize the AV object
    if voidP(av_idx) then
      timeline.type = #internal
    else
      initializeAV(av_idx)
    end if

  end if



  -- MISSING CODE HERE
  -- should test for different camera position (perhaps in
the attributes section)
  -- (REASON: when returning from a sub-scene, camera should
be in the same position
  -- as when the user left.  The "attributes" could hold
such positions.)
  -- MISSING CODE HERE

  -- set currenttime to start time
  timeline.current = timeline.start

  -- set the loopstart
  set loopstarthold = the milliseconds
  set timehold = 0
  -- turn on the stepframe loop
  sprite(sceneLayoutProps.spritenum).pInitialized = TRUE

  sendStatus("Timeline Initialized")

end




on commandLoopCall
  -- The Command Loop Call is the timeout handler for the
command loop
  -- It is called every 40ms (or 25 frames per second), and
executes
  -- commands in it's queue loop
  --
  -- STEPS:
  -- 1.  set the current time based on timeline source
(internal or external)
```

```
  -- 2.  if user-navigation is enabled, check for navigation
events
  -- 3.  if position-events are enabled, check for position
events
  -- 4.  check the instruction sets for new events to
include in the event queue
  -- 5.  execute items in the event queue


  -- set the current time based on internal or external
sources
  case timeline.type of
    #internal:
      timeline.current = ((the milliseconds) -
loopstarthold)
    #external:
      -- timeline.current = ((the milliseconds) -
loopstarthold)
      if sprite(av_object).mediaStatus = #playing then
        if ((the milliseconds - loopstarthold) > 2000) then
          loopstarthold = the milliseconds
          timehold = sprite(av_object).currentTime
          timeline.current = timehold
          -- put "get time" && timehold
        else
          timeline.current = timehold + (the milliseconds -
loopstarthold)
          -- put timeline.current
        end if
      end if

  end case

  -- check to see if user navigation is currently enabled
  if (timeline.navigation) then
    -- check for navigation actions (keypress & mouselooks)
    checkNavigationEvents()
  end if

  -- check to see if user position calls are currently
enabled
  if (timeline.position) then
    -- check user position against target areas
  end if

  -- if there are highlighted events,
  if (model_highlight_list.count <> 0) then
    -- cycle the color
    cycleModelHighlight(timeline.current)
  end if



  -- check for new event queues
  -- execute events in the event queue
  doQueuedEvents()

  -- check for proximity events
  if (prox_trigger_list.count > 0) then
    evaluateProximities()
  end if

  -- set directionally targetted objects
  updateObjectTargetting()

end
```

Navigation / Camera Scripts

```
global nav_settings
global currentscene
global user_keymap
global user_speedsettings
global scene
global cameraobject
global sceneLayoutProps
global default_fade_image
global camera_orientation

on resetNavSettings
  -- the Navigational Settings holds the various settings 
for keymapping
  -- and mouse look / navigate properties
  --
  -- #type holds the current type of navigation
  --  (examples: #free, #rotatecamera, #rotate object
  --
  -- #mouse holds the mouse look properties, as well as the 
mouse speed,
  --   invert mouse, and mouse sensitivity settings
  --
  -- #keymap holds the keymappings for navigation
  --
  -- #speed holds the movement & rotational speed settings

  nav_settings = [#type:[:], #mouse:[:], #keymap:[:], 
#speed:[:], #restrictions:[:]]

  -- update the navigational settings
  nav_settings.type = currentscene.settings.navigation.type 
-- navigational type
  nav_settings.keymap = user_keymap   -- directional & 
rotational keymap
  nav_settings.mouse = 
currentscene.settings.navigation.mouse   -- mouse look & 
sensitivity buffer settings
  nav_settings.speed = user_speedsettings  -- rotational & 
movement speeds
  nav_settings.restrictions = 
currentscene.settings.navigation.restrictions   -- axis 
rotation & positional restrictions

  -- if the mouse object is enabled, allow the 3d display 
object to receive mouse events
  if (nav_settings.mouse.enabled) then
    -- enable the mouse look functionality
    sprite(sceneLayoutProps.spritenum).pMouseLookEnabled = 
TRUE
```

```
    -- set the buffer zone for the sprite
    set centerpoint = point(sceneLayoutProps.x +
(sceneLayoutProps.w / 2), sceneLayoutProps.y +
(sceneLayoutProps.h / 2))

    set buf_p = (float(nav_settings.mouse.buffer_percent) /
200)

    set buf_x1 = integer(centerpoint[1] -
(sceneLayoutProps.w * buf_p))
    set buf_y1 = integer(centerpoint[2] -
(sceneLayoutProps.h * buf_p))
    set buf_x2 = integer(centerpoint[1] +
(sceneLayoutProps.w * buf_p))
    set buf_y2 = integer(centerpoint[2] +
(sceneLayoutProps.h * buf_p))

    sprite(sceneLayoutProps.spritenum).pMouseBufferZone =
rect(buf_x1, buf_y1, buf_x2, buf_y2)

    -- set the mouse speed & invert properties
    sprite(sceneLayoutProps.spritenum).pMouseSpeed =
user_speedsettings.mouse_speed
    sprite(sceneLayoutProps.spritenum).pMouseInvert =
user_speedsettings.mouse_invert

    -- set the sprite rect
    sprite(sceneLayoutProps.spritenum).pSpriteRect =
sprite(sceneLayoutProps.spritenum).rect
  end if


end

on checkNavigationEvents
  -- based on the navigation types & settings, this script
moves the camera
  -- (or selected object) appropriately, allowing user
manipulation of the camera
  -- throughout the model, or of an object within the model


  case (nav_settings.type) of

    #free:
      -- this is the free exploration camera
      set units = nav_settings.speed.move
      set degrees = nav_settings.speed.rotate


      -- move left
      if
(keypressed(numtoChar(nav_settings.keymap.move_left))) then
        cameraobject.translate(-units, 0, 0, #self)
      end if

      -- move right
      if
(keypressed(numtoChar(nav_settings.keymap.move_right))) then
        cameraobject.translate(units, 0, 0, #self)
      end if

      -- move forward
      if
(keypressed(numtoChar(nav_settings.keymap.move_forward)))
then
        -- check for z-axis restriction
        if (nav_settings.restrictions.zaxis) then
          -- z-axis is restricted, move the camera object
only
          cameraobject.translate(0,0,-units, #self)
        else
          -- z-axis is not restricted, rotate, move, then
unrotate the camera object
          set zangle = scene.camera[1].transform.rotation.x
          cameraobject.rotate(zangle,0,0, #self)
          cameraobject.translate(0,0,-units, #self)
          cameraobject.rotate(-zangle,0,0, #self)
        end if

      end if

      -- move backward
      if
(keypressed(numtoChar(nav_settings.keymap.move_backward)))
then
        -- check for z-axis restriction
        if (nav_settings.restrictions.zaxis) then
          -- z-axis is restricted, move the camera object
only
          cameraobject.translate(0,0,units, #self)
        else
          -- z-axis is not restricted, rotate, move, then
unrotate the camera object
          set zangle = scene.camera[1].transform.rotation.x
          cameraobject.rotate(zangle,0,0, #self)
          cameraobject.translate(0,0,units, #self)
          cameraobject.rotate(-zangle,0,0, #self)
        end if

      end if

      -- turn left
      if
(keypressed(numtoChar(nav_settings.keymap.rotate_left)))
then
        cameraobject.rotate(0, degrees, 0, #self)
      end if
      -- turn right
      if
(keypressed(numtoChar(nav_settings.keymap.rotate_right)))
then
        cameraobject.rotate(0, -degrees, 0, #self)
      end if

      -- turn up
      if
(keypressed(numtoChar(nav_settings.keymap.rotate_up))) then
        if ((scene.camera[1].transform.rotation.x + degrees)
< 90) then
          scene.camera[1].rotate(degrees, 0, 0)
        end if
      end if

      -- turn down
      if
(keypressed(numtoChar(nav_settings.keymap.rotate_down)))
then
        if ((scene.camera[1].transform.rotation.x - degrees)
> -90) then
          scene.camera[1].rotate(-degrees, 0, 0)
        end if
      end if



    #rotatecamera:
      -- camera is rotating /zooming about an object

    #rotateobject:
      -- an object is being rotated

    otherwise:
      -- nothing

  end case


end


on createCameraObject

  -- this creates the camera sphere which is used for
positioning, rotation,
  -- and collision detection of the camera

  -- NOTE FOR FURTHER DEVELOPMENT
  -- maybe allow user to define camera object size?
  -- would allow to limit motion / slipping through
cracks...
```

```
  -- allow multiple cameras?
  -- allow orthographic projection?

  -- CODE MISSING HERE
  -- check for an existing camera object and delete it if
found
  -- CODE MISSING HERE

  -- create the sphere
  -- sphere_mr = scene.newModelResource("mrsphere", #sphere,
#front)
  -- sphere_mr.radius = 5
  -- cameraobject = scene.newModel("camera_object",
sphere_mr)
  -- cameraobject.transform.position = vector(0,0,0)
  -- cameraobject.transform.rotation = vector(0,0,0)

  -- create the box
  box_mr = scene.newModelResource("mrbox", #box, #front)
  box_mr.length = 2
  box_mr.height = 2
  box_mr.width = 2
  cameraobject = scene.newModel("camera_object", box_mr)
  cameraobject.visibility = #none
  cameraobject.transform.position = vector(0,0,0)
  cameraobject.transform.rotation = vector(0,0,0)

  -- position the camera
  scene.camera[1].transform.position = vector(0,0,0)
  scene.camera[1].transform.rotation = vector(0,0,0)

  -- make the sphere the parent of the camera
  scene.camera[1].parent = cameraobject

  -- if there is a headlight object, activate that
  if (currentscene.settings.camera.headlight.enabled) then
    turnOnCameraHeadlight()
  end if



end


on setupCameraLocation
  -- sets the initial position / orientation

  -- load the position settings
  set cameraSettings =
currentscene.settings.camera.initial_position

  -- set the identity transform
  set cameraTransform = transform()
  -- position the camera
  cameraTransform.position = vector(cameraSettings.x_pos,
cameraSettings.y_pos, cameraSettings.z_pos)

  -- pan the camera object
  cameraTransform.rotation = vector(0,
cameraSettings.pan_deg, 0)

  -- update the camera object
  cameraobject.transform = cameraTransform

  -- tilt the camera
  scene.camera[1].rotate(cameraSettings.tilt_deg, 0, 0)

  -- reset the camera projection
  scene.camera[1].projection = #perspective

  -- set the camera field of view
  -- 8/18/05
  -- LEAVE OUT FIELD OF VIEW FOR NOW
  -- scene.camera[1].fieldofview =
cameraSettings.fieldofview


end

on createCameraTextures()
  -- creates an images for the default fade color
  -- "custom fade" images should already be members

  -- clear the exisiting overlays  (just in case)
  set i = scene.camera[1].overlay.count
  repeat while i > 0
    scene.camera[1].removeOverlay[i]
    i = i + 1
  end repeat


  -- create the image
  default_fade_image = image(2,2, 16) -- 2 x 2 because we
only need a color fill
  default_fade_image.fill(default_fade_image.rect,
rgb(currentscene.settings.camera.fading.default_color))

  -- add the texture to the scene
  scene.newTexture("default_fade_texture", #fromImageObject,
default_fade_image)

  -- add the overlay to the camera (initially off)

scene.camera[1].addOverlay(scene.texture("default_fade_textu
re"), point(0,0), 0)

  -- get the scale:
```

```
  if (sceneLayoutProps.w > sceneLayoutProps.h) then
    set overlay_scale = sceneLayoutProps.w / 2.0
  else
    set overlay_scale = sceneLayoutProps.h / 2.0
  end if

  -- set the scale
  scene.camera[1].overlay[1].scale = overlay_scale

  -- turn off the overlay (will turn it back on if needed)
  scene.camera[1].overlay[1].blend = 0

  -- CODE MISSING HERE!!!
  -- Import the custom fade images as textures
  -- catalog the index source of these items / position as
necessary.

end

on cameraFade fade_atts, fade_percent
  -- fades the camera "in" or "out" by altering the blend on
the camera overlay

  -- determine which texture to fade
  if (fade_atts.overlay = #default) then
    set this_overlay = 1
  else
    -- get the custom event
    -- MISSING CODE HERE!!!
  end if

  -- determine which direction:
  if (fade_atts.direction = #in) then
    -- fade in
    set percentage = integer(100 - (fade_percent * 100))
  else
    -- fade out
    set percentage = integer(fade_percent * 100)
  end if

  scene.camera[1].overlay[this_overlay].blend = percentage

  if (fade_percent = 1) then
    return TRUE
  end if

end

on turnOnCameraHeadlight
  -- creates a headlight for the camera, with a light gray
light (avoid bright white)

  if (currentscene.settings.camera.headlight.type = #spot)
then
```

```
    set camera_angle = scene.camera[1].fieldOfView

    scene.newLight("camera_headlight", #spot)

    -- move this light to the camera
    scene.light("camera_headlight").transform =
scene.camera[1].getWorldTransform()

    -- set the camera as the parent object
    scene.light("camera_headlight").parent = scene.camera[1]

    -- move the headlight back
    scene.light("camera_headlight").translate(0,0,1)
    -- set the spot color
    scene.light("camera_headlight").color =
currentscene.settings.camera.headlight.color

    -- set the spot angle
    scene.light("camera_headlight").spotAngle = camera_angle
/ 1.5
    scene.light("camera_headlight").spotDecay = TRUE
  else if (currentscene.settings.camera.headlight.type =
#directional) then

    scene.newLight("camera_headlight", #directional)

    -- set the rotation to that of the camera
    scene.light("camera_headlight").transform =
scene.camera[1].getWorldTransform()

    -- set the camera as the parent object
    scene.light("camera_headlight").parent = scene.camera[1]

    -- set the headlight color
    scene.light("camera_headlight").color =
currentscene.settings.camera.headlight.color
  end if

end

on getCameraPosition
  -- gets the current position of the camera object
  set this_vec = cameraobject.transform.position
  return this_vec
end

on getPanDirection
  -- gets the overall facing direction of a camera
  set zangle = getCameraTilt()
  set position1 = getCameraPosition()

  cameraobject.translate(0,0,-1, #self)
  set position2 = getCameraPosition()
  cameraobject.translate(0,0,1, #self)
  return position2 - position1
end

on getPanAngle
  -- returns the angle of the pan, based on a vector of
(1,0,0)  (+x axis)
  set panDir = getPanDirection()
  set pan_angle = vector(0,0,-1).angleBetween(panDir)

  set clockwise = vector(-1,0,0).angleBetween(panDir)
  set counterclockwise = vector(1,0,0).angleBetween(panDir)

  if counterclockwise < clockwise then
    -- negative angle
    pan_angle = -1 * pan_angle
  end if
  return pan_angle
end


on getCameraTilt
  -- returns the tilt angle of the camera
  set this_angle = scene.camera[1].transform.rotation.x
  return this_angle
end

on getCameraDirection
  -- gets the overall facing direction of a camera
  set zangle = getCameraTilt()
  set position1 = getCameraPosition()
  cameraobject.rotate(zangle,0,0, #self)
  cameraobject.translate(0,0,-1, #self)
  set position2 = getCameraPosition()
  cameraobject.translate(0,0,1, #self)
  cameraobject.rotate(-zangle,0,0, #self)
  return position2 - position1
end

on setCameraPosition camera_position
  -- moves the camera to a new position
  cameraobject.transform.position = camera_position
end

on translateCameraWorld translation_vector
  -- translates the camera by a vector
  cameraobject.translate(translation_vector, #world)
end

on tiltCameraAbsolute tilt_angle
  -- tilts the camera angle
  if (tilt_angle <= 90) AND (tilt_angle >= -90) then
    scene.camera[1].transform.rotation.x = tilt_angle
  end if
```

```
end

on panCameraRelative pan_angle
  -- pans the camera this many degrees
  cameraobject.rotate(0, -pan_angle, 0, #self)
end


on tiltCameraRelative tilt_angle
  -- tilts the camera by the specified value, or to maximum
angle
  set current_angle = scene.camera[1].transform.rotation.x
  set new_angle = current_angle + tilt_angle

  if (new_angle > 90.0) then
    new_angle = 90
  else if (new_angle < -90) then
    new_angle = -90
  end if

  scene.camera[1].transform.rotation.x = new_angle
end


on getPanAndTiltDifference target_position
  -- returns the pan and tilt angles to a position

  -- pan angle
  set current_vector = getPanDirection()
  set current_position = getCameraPosition()
  set pan_vector = vector(target_position.x -
current_position.x, 0, target_position.z -
current_position.z)

  set pan_angle = current_vector.angleBetween(pan_vector)

  -- determine if this is clockwise or counter-clockwise
from vector
  set check_trans = transform()
  check_trans.position = current_vector
  check_trans.rotate(vector(0,0,0), vector(0,1,0), 90)
  set clockwise =
(check_trans.position).angleBetween(pan_vector)
  check_trans.rotate(vector(0,0,0), vector(0,1,0), 180)
  set counterclockwise =
(check_trans.position).angleBetween(pan_vector)

  -- whichever side has the smaller angle is the direction
  if (counterclockwise > clockwise) then
    pan_angle = -1 * pan_angle
  end if

  -- tilt angle
  set tilt_position = target_position - current_position
  set tilt_from_plane =
(tilt_position).angleBetween(pan_vector)

  -- negative angle check
  if (tilt_position.y < 0) then
    tilt_from_plane = -1 * tilt_from_plane
  end if

  set current_tilt = getCameraTilt()
  set tilt_angle = tilt_from_plane - current_tilt

  return [#pan: pan_angle, #tilt: tilt_angle]

end

on pointCameraAtPosition target_position
  -- targets the camera at a given position vector

  -- get the difference
  set angle_difference =
getPanAndTiltDifference(target_position)

  -- pan & tilt to match
  panCameraRelative(angle_difference.pan)
  tiltCameraRelative(angle_difference.tilt)
end

on pointCameraAtDirection target_direction
  -- targets the camera to a given direction
  set current_pos = getCameraPosition()
  set new_position = current_pos + target_direction

  pointCameraAtPosition(new_position)
end

on pointCameraAtPathDirection target_direction
  -- targets the camera to a given direction
  set current_pos = getCameraPosition()
  set new_position = current_pos + target_direction

  pointCameraAtPosition(new_position)
  panCameraRelative(camera_orientation)

end

on rotateCameraByAngle rotation_axis, rotation_angle
  -- rotates the camera by an angle about an axis, relative
to the current direction
  set current_dir = getCameraDirection()
  set dir_hold = transform()

  dir_hold.position = current_dir -- set the relative
direction as the position
```

```
  dir_hold.rotate(vector(0,0,0), rotation_axis,
rotation_angle) -- rotate the vector

  new_dir = dir_hold.position -- get the new direction
  pointCameraAtDirection(new_dir) -- point it at the new
direction

end


on rotateCameraFromDirection pan_angle
  -- this only effects the pan angle - there is no axis of
rotation other than "up"

  -- rotate the camera
  panCameraRelative(pan_angle)

  -- counter-rotate the direction
  camera_orientation = camera_orientation + (-1 * pan_angle)

end

on resetCameraOrientation
  -- resets the camera point-at-orientation to it's original
value
  camera_orientation = 0
end
```

DisplayObject Behavior

```
-- 3D Display Object Behavior scripts
-- Began 1/20/03
--
-- These scripts will handle item specific calls from and
functions of
-- the Shockwave 3D element, including the initialization
and positioning,
-- the step-frame event (which fuels the internal timeline &
event queue)
-- and various mouse events (click-ons, etc...)


property pInitialized
property pMouseLookEnabled
property pMouseLookOn
property pMouseBufferZone
property pMouseInvert
property pMouseSpeed
property pSpriteRect



-- BEGIN SPRITE  -------------------------------------------
-------------
--
-- set the initial properties for the scene object
on beginSprite me
  global sceneLayoutProps

  -- add this to the actor list so that it will receive a
stepframe event
  add the actorlist, me

  -- set the initial values
  pInitialized = FALSE
  pMouseLookEnabled = FALSE
  pMouseLookOn = FALSE
  pMouseBufferZone = rect(0,0,0,0)
  pMouseSpeed = 1
  pMouseInvert = 0
  pSpriteRect = rect(0,0,0,0)

  -- initialize this sprite by reporting the sprite number
into the scene properties
  setScenePosition(me.spritenum)

end beginSprite
------------------------------------------------------------
-------------
```

```
-- LOOPING CALL -------------------------------------------
-------------
-- Once initialized, this will execute the command loop at
each frame step
-- event encountered by the scene object

on stepFrame me

  if (pInitialized) then
    commandLoopCall()
  end if

end
------------------------------------------------------------
-------------



-- MOUSE INTERACTION SCRIPTS -------------------------------
-------------
-- Once initialized, this will check for mouse events, as
well as the
-- current state of the mouse, and will execute camera
events accordingly

on keyDown me
  -- checks for the mouselook on/off event
  global user_keymap

  if (pMouseLookEnabled) then
    -- only look for this event if this is enabled
    if charToNum(the key) = user_keymap.mouse_look then
      if (pMouseLookOn) then
        pMouseLookOn = FALSE
      else
        pMouseLookOn = TRUE
      end if
    end if
  else
    pass
  end if

end

on mouseWithin me
  global nav_settings
  global sceneLayoutProps
  global scene
  global cameraobject

  if (pMouseLookEnabled) then
    if (pMouseLookOn) then
```

```
      -- check the position of the mouse against the buffer
zone
      if NOT(inside(the mouseloc, pMouseBufferZone)) then

        -- determine the position of the mouse
        set curloc = the mouseloc

        -- pan test & execute
        if (curloc[1] < pMouseBufferZone[1]) then
          -- pan left
          set x_percent = (float(pMouseBufferZone[1] -
curloc[1]) / (pMouseBufferZone[1] - pSpriteRect[1]))
          set x_angle = (pMouseSpeed * x_percent)
          cameraobject.rotate(0, x_angle, 0, #self)

        else if (curloc[1] > pMouseBufferZone[3]) then
          -- pan right
          set x_percent = (float(curloc[1] -
pMouseBufferZone[3]) / (pSpriteRect[3] -
pMouseBufferZone[3]))
          set x_angle = -(pMouseSpeed * x_percent)
          cameraobject.rotate(0, x_angle, 0, #self)
        end if

        -- tilt test & execute
        if (curloc[2] < pMouseBufferZone[2]) then
          -- tilt up (or down if inverted)
          set y_percent = (float(pMouseBufferZone[2] -
curloc[2]) / (pMouseBufferZone[2] - pSpriteRect[2]))
          set y_angle = (pMouseSpeed * y_percent)
          if (pMouseInvert) then
            y_angle = -(y_angle)
            if ((scene.camera[1].transform.rotation.x +
y_angle) > -90) then
              scene.camera[1].rotate(y_angle, 0, 0)
            else
              set max_angle = -90 -
scene.camera[1].transform.rotation.x -- + 1
              scene.camera[1].rotate(max_angle, 0, 0)
            end if
          else
            if ((scene.camera[1].transform.rotation.x +
y_angle) < 90) then
              scene.camera[1].rotate(y_angle, 0, 0)
            else
              set max_angle = 90 -
scene.camera[1].transform.rotation.x -- - 1
              scene.camera[1].rotate(max_angle, 0, 0)
            end if
          end if

        else if (curloc[2] > pMouseBufferZone[4]) then
          -- tilt down (or up if inverted)
          set y_percent = (float(curloc[2] -
pMouseBufferZone[4]) / (pSpriteRect[4] -
pMouseBufferZone[4]))
          set y_angle = -(pMouseSpeed * y_percent)
          if (pMouseInvert) then
            y_angle = -(y_angle)
            if ((scene.camera[1].transform.rotation.x +
y_angle) < 90) then
              scene.camera[1].rotate(y_angle, 0, 0)
            else
              set max_angle = 90 -
scene.camera[1].transform.rotation.x -- - 1
              scene.camera[1].rotate(max_angle, 0, 0)
            end if
          else
            if ((scene.camera[1].transform.rotation.x +
y_angle) > -90) then
              scene.camera[1].rotate(y_angle, 0, 0)
            else
              set max_angle = -90 -
scene.camera[1].transform.rotation.x -- + 1
              scene.camera[1].rotate(max_angle, 0, 0)
            end if
          end if
        end if

      end if

    end if
  end if

end


on mouseUp me
  -- registers the mouse click to recognize potential mouse
clicks
  set m = VOID
  pt = the mouseLoc - point(sprite(me.spritenum).left,
sprite(me.spritenum).top)

  checkClickPoint(pt)

end


------------------------------------------------------------
-------------
-- BEGIN SPRITE  -------------------------------------------
-------------
```

LOADFILE2

```
global scene          -- we are going to reference the 3D
castmember at least once in this script
global currentscene
global scenemodels



on exitFrame me
  put scene.state

  if check3Dready(scene) then    -- call custom handler:  is
the SW3D castmember ready?

    scene.resetworld()
    --
scene.loadFile(getExternalModelpath(currentscene.scenenum))

scene.loadFile(getExternalModelpath(getNamedListIndex(scenem
odels, currentscene.scenename)))
    sendStatus("model loaded")
    go "3dinit"                    -- if it is ready, go to
the "init" marker
  else
    go to the frame                  -- if it is not ready,
wait here
  end if
end
```

3D RERUN

```
global diagnostic

on exitframe me

  -- check for the diagnostics
  if (diagnostic) then
    getCameraStats()
  end if


  -- loop on this frame
  go to "3dRun"

end
```

3D SPRITE SETUP

```
global sceneLayoutProps
global fullstatus

on exitframe me

  -- puppetSprite(sceneLayoutProps.spritenum, TRUE)


  -- ready to move onto the next session
  fullstatus = #ready

end
```

3D DISPLAY VERIFICATION

```
global fullstatus

on exitFrame me

  if fullstatus = #ready then
    go to "3dhold"
  else
    go to the frame
  end if
end
```

Layout Scripts

```
global sceneLayoutProps

on setScenePosition scene_spritenum

  sceneLayoutProps.spritenum = scene_spritenum
  sprite(scene_spritenum).rect = rect(sceneLayoutProps.x,
sceneLayoutProps.y, (sceneLayoutProps.w +
sceneLayoutProps.x), (sceneLayoutProps.h +
sceneLayoutProps.y))

end
```

Current Scene Setup

```
global scene
global currentscene

on setWorldParameters
  -- sets the overall scene parameters, i.e. World-
Translation, Hierarchy,
  -- Background, Directional and AmbientLights

  -- setup the global model / group / light catalogs
  catalogModelNames()
  catalogGroupNames()
  catalogLightNames()

  -- preserve or reset the inherent parent / child
relationships
  if
(currentscene.settings.model.worldsettings.structure.preserv
e_structure = FALSE) then
    resetInherentStructure()
  end if

  -- set the world orientation
  resetWorldOrientation()

  -- set the world lighting properties
  resetWorldLighting()


end



on resetWorldLighting
  -- sets the bgcolor, ambient color, etc of the model.
  -- also turns off the director directional preset
  scene.bgcolor =
rgb(currentscene.settings.lighting.worldsettings.background_
color)
  scene.ambientColor =
rgb(currentscene.settings.lighting.worldsettings.ambient_col
or)
end


on resetWorldOrientation
  -- creates a temporary group clone of World Group, then
moves/rotates/scales
  -- as determined by the orientation

  sendStatus("Setting world orientation")
  -- clone the World group
  createGroupClone("World", "prescene_Transform")

  -- set the variables
  set change_position =
currentscene.settings.model.worldsettings.orientation.positi
on
  set change_rotation =
currentscene.settings.model.worldsettings.orientation.rotati
on
  set change_scale =
currentscene.settings.model.worldsettings.orientation.scale

  set groupTransform = transform()

  groupTransform.position = change_position
  groupTransform.rotation = change_rotation
  groupTransform.scale = change_scale

  -- set the group to the new transform
  scene.group("prescene_Transform").transform =
groupTransform

  -- remove the group
  destroyGroup("prescene_Transform")


end
```

Interpolation Scripts

```
-- these scripts interpolate between two given values and a
-- corresponding (float) percentage.


on interpolateColor colorA, colorB, percentage
  -- interpolates between two rgb formatted colors, given a
decimal percentage

  -- set the intermedite object
  set colorC = colorA * (1 - percentage)
  set colorD = colorB * percentage

  set return_color = colorC + colorD

  return return_color

end
```

Lighting Interaction Scripts

```
global scene
global current_light_list
global off_light_list
global light_orientation_list

on catalogLightNames
  -- populates the CURRENT LIGHT LIST with the name of each
light within world
  current_light_list = []
  repeat with i = 1 to scene.light.count
    current_light_list.append(scene.light[i].name)
  end repeat
end

on verifyLight light_name
  -- verifies the existance of a light within the world
  if (current_light_list.getOne(light_name)) then
    return TRUE
  else
    return FALSE
  end if
end

on getLightPosition light_name
  -- verifies that the light exists, and if so, returns the
position as a vector
  if (verifyLight(light_name)) then
    set this_vec =
scene.light(light_name).transform.position
    return this_vec
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on getLightRotation light_name
  -- verifies that the light exists, and if so, returns the
rotation.
  if (verifyLight(light_name)) then
    set this_vec =
scene.light(light_name).transform.rotation
    return this_vec
  else
    return -1   -- denotes the model doesn't exist
  end if
end

on getLightDirection light_name
  -- verifies that the light exists, and if so, returns the
direction
  if (verifyLight(light_name)) then
```

```
    set this_direction =
scene.light(light_name).pointAtOrientation[1]
    set this_axisangle =
scene.light(light_name).getWorldTransform().axisAngle

    set world_direction =
rotateDirectionAboutAxis(this_direction, this_axisangle[1],
this_axisangle[2])

    return world_direction

  end if
end

on setLightPosition light_name, light_position
  -- verifies that light exists, if so moves it to the new
position vector
  if (verifyLight(light_name)) then
    set this_trans =
scene.light(light_name).getWorldTransform().position
    set translate_vec = light_position - this_trans
    scene.light(light_name).translate(translate_vec, #world)
  else
    return -1  -- denotes the light doesn't exist
  end if

end


on setLightSelfRotation light_name, rotation_axis,
rotation_angle
  -- verifies that the light exists, if so rotates it about
the self-axis
  if (verifyLight(light_name)) then
    scene.light(light_name).rotate(vector(0,0,0),
rotation_axis, rotation_angle, #self)
  else
    return -1  -- denotes the light doesn't exist
  end if
end

on setLightWorldRotation light_name, rotation_axis,
rotation_angle
  -- verifies that the light exists, and if so, rotates it
about the world defined axis
  -- don't need to check the axis for a 0 value, because
director ignores this command
  if (verifyLight(light_name)) then
    set posVec =
scene.light(light_name).getWorldTransform().position
    scene.light(light_name).rotate(posVec, rotation_axis,
rotation_angle, #world)
  else
    return -1  -- denotes the light doesn't exist
  end if

end

on pointLightAtPosition light_name, target_position
  -- verifies that light exists, if so changes the rotation
to match the new vector
  if (verifyLight(light_name)) then
    -- check for an invalid position (identical to current
point)
    if (target_position <>
scene.light(light_name).getWorldTransform().position) then
      scene.light(light_name).pointAt(target_position)
    end if
  else
    return -1 -- denotes the light doesn't exist
  end if
end

on pointLightAtDirection light_name, target_direction
  -- verifies that light exists, if so changes the rotation
to match the new vector
  if (verifyLight(light_name)) then
    -- check for an invalid position (identical to current
point)
    set light_position =
scene.light(light_name).getWorldTransform().position
    set target_position = light_position + target_direction
    if vectorP(target_direction) then
      if (target_direction.magnitude <> 0) then
        scene.light(light_name).pointAt(target_position)
      end if
    end if

  else
    return -1 -- denotes the light doesn't exist
  end if

end

on rotateLightFromDirection light_name, axis_vector, angle
  -- verifies that the light exists, if so rotates it, but
counter rotates the point at direction
  if (verifyLight(light_name)) then
    -- rotate the light
    scene.light(light_name).rotate(vector(0,0,0),
axis_vector, angle, #self)
    -- counter-rotate the direction
    rotateLightOrientation(light_name, axis_vector, (-1 *
angle))
  else
    return -1  -- denotes the light doesn't exist
  end if
```

```
end


on rotateLightOrientation light_name, axis_vector, angle
  -- verifies the light exists, then rotates around the axis
vector by angle (degrees)
  if (verifyLight(light_name)) then
    -- is this in the light_orientation_list?
    set this_orientation =
light_orientation_list.getaProp(light_name)

    set dir_vec =
scene.light(light_name).pointAtOrientation[1]
    set up_vec =
scene.light(light_name).pointAtOrientation[2]

    if voidP(this_orientation) then
      -- set the orientation hold values
      light_orientation_list.addProp(light_name, [dir_vec,
up_vec])
    end if

    -- set up the transforms
    set dir_trans = transform()
    set up_trans = transform()

    dir_trans.position = dir_vec
    up_trans.position = up_vec

    -- rotate about the given axis
    dir_trans.rotate(vector(0,0,0), axis_vector, angle)
    up_trans.rotate(vector(0,0,0), axis_vector, angle)

    -- apply the new values
    scene.light(light_name).pointAtOrientation =
[dir_trans.position, up_trans.position]

  else
    return -1  -- denotes the light doesn't exist
  end if
end

on resetLightOrientation light_name
  -- verifies that the light exists, then resets to the
original values
  if (verifyLight(light_name)) then
    set this_orientation =
light_orientation_list.getaProp(light_name)
    if NOT(voidP(this_orientation)) then
      scene.light(light_name).pointAtOrientation =
this_orientation
      light_orientation_list.deleteProp(light_name)
    end if
  else
    return -1
  end if

end



on turnOffLight light_name
  -- verify that the light currently exists
  if (verifyLight(light_name)) then
    -- build the off_light_list property list
    set light_props = [#type: VOID, #color: VOID,
#transform: VOID, #specular: VOID, #spotangle: VOID,
#spotdecay: VOID, #attenuation: VOID]
    set this_light = scene.light(light_name)

    light_props.type = this_light.type
    light_props.color = this_light.color

    if (this_light.type <> #ambient) then
      -- get the world position / rotation
      light_props.transform = this_light.getWorldTransform()
      -- get other properties
      light_props.specular = this_light.specular
      if (this_light.type <> #directional) then
        light_props.attenuation = this_light.attenuation
        if (this_light.type = #spot) then
          light_props.spotangle = this_light.spotangle
          light_props.spotdecay = this_light.spotdecay
        end if
      end if
    end if

    -- add an attribute (light name) with the value
(property list) to the OFF LIGHT LIST
    off_light_list.addProp(light_name, light_props)

    -- delete the light
    scene.deleteLight(light_name)

    -- update the current light list
    catalogLightNames()

    return TRUE

  else
    return 0  -- denotes that light doesn't exist, or is
already off
  end if
end


on turnOnLight light_name
```

```
  -- verifies that the light has been turned off (is in the
OFF LIGHT LIST)
  set light_props = off_light_list.getaProp(light_name)

  if NOT(voidP(light_props)) then
    -- the light is turned off.  create a new light with
these properties
    scene.newLight(light_name, light_props.type)

    scene.light(light_name).color = light_props.color

    if (light_props.type <> #ambient) then

      scene.light(light_name).transform =
light_props.transform
      scene.light(light_name).specular =
light_props.specular

      if (light_props.type <> #directional) then
        scene.light(light_name).attenuation =
light_props.attenuation
        if (light_props.type = #spot) then
          scene.light(light_name).spotangle =
light_props.spotangle
          scene.light(light_name).spotdecay =
light_props.spotdecay
        end if

      end if
    end if

    -- recatalog the light names
    catalogLightNames()

    -- remove this item from the OFF LIGHT LIST
    off_light_list.deleteProp(light_name)

    return TRUE

  else
    return 0  -- denotes that light doesn't exist, or is
already on
  end if


end


on changeLightColor event_attributes, change_percent
  -- changes the color of a light by a percentage
  --
  -- event attributes are:
  -- #light_name: <light_name> OR #background (ambient light
should have a name)
```

```
  -- #start_value: <rgb color> OR #this
  -- #end_value: <rgb color>
  --
  -- SHOULD I INCLUDE SCRIPTING FOR IF THE LIGHT IS
CURRENTLY TURNED OFF?

  if (event_attributes.light_name = #background) then
    -- change the background color of the scene
    if (event_attributes.start_value = #this) then
      event_attributes.start_value = scene.bgcolor
    end if

    -- set the current scene color
    scene.bgcolor =
interpolateColor(event_attributes.start_value,
event_attributes.end_value, change_percent)

    if (change_percent = 1) then
      return TRUE
    else
      return event_attributes
    end if

  else

    if (verifyLight(event_attributes.light_name)) then
      -- light change scripts here.
      if (event_attributes.start_value = #this) then
        event_attributes.start_value =
scene.light(event_attributes.light_name).color
      end if

      -- set the interpolated color of the light
      scene.light(event_attributes.light_name).color =
interpolateColor(event_attributes.start_value,
event_attributes.end_value, change_percent)

      if (change_percent = 1) then
        return TRUE
      else
        return event_attributes
      end if

    else if
NOT(voidP(off_light_list.getaProp(event_attributes.light_nam
e))) then
      -- light is turned off, so just change the light's
properties in the off_light_list
      off_light_list[event_attributes.light_name].color =
event_attributes.end_value

      return TRUE

    else
```

```
      return -1  -- denotes that this light doesn't exist,
cancel the command
    end if

  end if
end

on fadeOutLight light_name, fade_duration
  -- verifies that this light exists,
  -- saves the current color
  -- fade out the light
  -- turn off the light
  -- update the color of the light in the off light list

  -- MISSING CODE HERE!!!
  -- Include code for determining appropriate start time.
  -- Right now, this only accepts the "#now" command

  if (verifyLight(light_name)) then
    -- light exists - get the properties
    set light_color = scene.light(light_name).color

    set fadeout_name = "fadeout_" & the milliseconds & "_1_"
& light_name
    set turnoff_name = "fadeout_" & the milliseconds & "_2_"
& light_name
    set reset_name = "fadeout_" & the milliseconds & "_3_" &
light_name


    -- set the fade out event
    addEventToQueue([#name: fadeout_name, #command:
#change_lightcolor, #starttime: #now, #duration:
fade_duration, #priority: #noverify,
#attributes:[#light_name: light_name, #start_value: #this,
#end_value: rgb(0,0,0)], #hold_trigger: #none])

    -- set the turn off event
    addEventToQueue([#name: turnoff_name, #command:
#toggle_light, #starttime: #hold, #duration: 0, #priority:
#verify, #attributes:[#light_name: light_name, #toggle:
#off], #hold_trigger: fadeout_name])

    -- reset the color
    addEventToQueue([#name: reset_name, #command:
#change_lightcolor, #starttime: #hold, #duration: 0,
#priority: #verify, #attributes:[#light_name: light_name,
#start_value: #this, #end_value: light_color],
#hold_trigger: turnoff_name])


  else
    return FALSE
  end if
```

```
end

on fadeInLight light_name, fade_duration
  -- verifies that this light exists,
  -- saves the final color from the OFF LIGHT LIST
  -- updates the off light
  -- turns on the light
  -- fades to the final color

  -- MISSING CODE HERE!!!
  -- Include code for determining appropriate start time.
  -- Right now, this only accepts the #now command

  set light_props = off_light_list.getaProp(light_name)

  if voidP(light_props) then
    return FALSE
  else

    set light_color = light_props.color

    set reset_name = "fadein_" & the milliseconds & "_1_" &
light_name
    set turnon_name = "fadein_" & the milliseconds & "_2_" &
light_name
    set fadein_name = "fadein_" & the milliseconds & "_3_" &
light_name

    -- reset the color
    addEventToQueue([#name: reset_name, #command:
#change_lightcolor, #starttime: #now, #duration: 0,
#priority: #verify, #attributes:[#light_name: light_name,
#start_value: #this, #end_value: rgb(0,0,0)], #hold_trigger:
#none])

    -- turn on the light
    addEventToQueue([#name: turnon_name, #command:
#toggle_light, #starttime: #hold, #duration: 0, #priority:
#verify, #attributes:[#light_name: light_name, #toggle:
#on], #hold_trigger: reset_name])

    -- set the fade in event
    addEventToQueue([#name: fadein_name, #command:
#change_lightcolor, #starttime: #hold, #duration:
fade_duration, #priority: #verify, #attributes:[#light_name:
light_name, #start_value: #this, #end_value: light_color],
#hold_trigger: turnon_name])

  end if
end
```

Shader Scripts

```
global scene
global currentscene
global custom_shader_list
global model_highlight_list
global model_opacity_list
global model_color_list
global invisible_model_list



on createCustomShader model_name
  -- verify that the model exists
  if (verifyModel(model_name)) then
    -- model exists
    if voidP(custom_shader_list.getaProp(model_name)) then

      -- deep clone this model
      set cloned_model_name = model_name & "_shadeclone"
      set cloned_model =
scene.model(model_name).clonedeep(cloned_model_name)

      -- turn the model visibility off
      set this_visibility =
scene.model(model_name).visibility
      scene.model(model_name).visibility = #none

      -- set the cloned model parent to the main model

scene.model(model_name).addChild(scene.model(cloned_model_na
me), #preserveWorld)

      -- get each shader attribute
      set shader_list = [:]

      repeat with shadecount = 1 to
scene.model(cloned_model_name).shaderList.count
        set this_shader =
scene.model(cloned_model_name).shaderList[shadecount]

        -- check to see if this is already in the shader
list
        if voidP(shader_list.getaProp(this_shader.name))
then
          -- if not, add the props to the shader list
          shader_list.addProp(this_shader.name, [#ambient:
this_shader.ambient, #diffuse: this_shader.diffuse, #blend:
this_shader.blend])
        end if

      end repeat
      -- create the stored property list
      set model_props = [#clone_name: cloned_model_name,
#shader:shader_list, #visibility: this_visibility]

      -- add the model to the custom_shader_list
      custom_shader_list.addProp(model_name, model_props)

      return TRUE  -- denotes successful completion

    else
      return 0  -- denotes model already displays custom
shader
    end if
  else
    return -1  -- denotes an error (model does not exist)
  end if
end


on removeCustomShader model_name
  -- verify that the model exists
  if (verifyModel(model_name)) then
    -- make sure that it is in the custom shader list
    set this_model = custom_shader_list.getaProp(model_name)

    if voidP(this_model) then
      return 0  -- denotes that this model doesn't have a
custom shader
    end if

    -- set the cloned item
    cloned_model_name = this_model.clone_name

    -- set the kill lists
    set resource_kill_name =
scene.model(cloned_model_name).resource.name
    set texture_kill_list = []

    -- kill the model
    scene.deleteModel(cloned_model_name)

    -- kill the resources
    scene.deleteModelResource(resource_kill_name)

    -- catalog the textures, and kill the shaders
    repeat with shade_count = 1 to this_model.shader.count
      this_shader =
scene.shader(this_model.shader.getPropAt(shade_count))

      -- catalog the textures
      repeat with texture_count = 1 to 8  -- check all of
the textures
        set this_texture =
this_shader.textureList[texture_count]
```

```
        -- does this already exist in the kill list?
        if NOT(voidP(this_texture)) then
          this_texture = this_texture.name
          if texture_kill_list.getOne(this_texture) = FALSE
then
            texture_kill_list.append(this_texture)
          end if
        end if

      end repeat

      -- kill the shader

scene.deleteShader(this_model.shader.getPropAt(shade_count))
    end repeat

    -- kill the textures
    repeat with texture_count = 1 to texture_kill_list.count
      scene.deleteTexture(texture_kill_list[texture_count])
    end repeat

    -- turn the model visibility on
    scene.model(model_name).visibility =
this_model.visibility

    -- remove this object from the custom shader list
    custom_shader_list.deleteProp(model_name)

    -- if this is in the MODEL HIGHLIGHT LIST, remove it as
well
    model_highlight_list.deleteProp(model_name)

    return TRUE  -- denotes successful completion

  else
    return -1  -- denotes that model doesn't exist
  end if


end


on highlightModel model_name, event_name
  -- highlights the specified model, and sets the optional
event trigger.

  -- MISSING CODE HERE
  -- check to see if trigger event exists

  -- check to see if the model exists
  if (verifyModel(model_name)) then
    set highlight_test =
model_highlight_list.getaProp(model_name)
```

```
    if voidP(highlight_test) then
      -- is not highlighted yet
      createCustomShader(model_name)

      -- add to the highlight list
      model_highlight_list.addProp(model_name,
[#trigger_event: event_name])

    else
      -- model is already in the highlight list
      -- change the event trigger to the current trigger

      -- MISSING CODE HERE
      -- current trigger alteration

      return TRUE
    end if

  else
    return -1  -- denotes that model does not exist
  end if

end

on unHighlightModel model_name
  -- returns colors, and removes from highlight list
  -- removes custom shader if not located elsewhere

  set highlight_check =
model_highlight_list.getaProp(model_name)

  if voidP(highlight_check) then
    -- this model isn't highlighted
    return FALSE
  else
    -- return the colors
    set this_model = custom_shader_list.getaProp(model_name)
    repeat with shader_count = 1 to this_model.shader.count
      set this_shader =
this_model.shader.getPropAt(shader_count)
      scene.shader(this_shader).diffuse =
this_model.shader[shader_count].diffuse
      scene.shader(this_shader).ambient =
this_model.shader[shader_count].ambient
    end repeat

    -- is this in other lists?
    if voidP(model_opacity_list.getaProp(model_name)) then
-- MISSING CODE HERE: ADD OTHER CASES (color / texture
changes)
      -- kill the custom shader
      removeCustomShader(model_name)
    end if
    -- remove from the list
    model_highlight_list.deleteProp(model_name)
  end if

end


on cycleModelHighlight current_time
  -- cycles the shaders of the highlighted objects

  set highlight_diffuse =
currentscene.settings.model.worldsettings.highlight.diffuse
  set highlight_ambient =
currentscene.settings.model.worldsettings.highlight.ambient

  set cycle_time =
currentscene.settings.model.worldsettings.highlight.cycle_du
ration
  set half_cycle = cycle_time / 2

  set cycle_position = current_time mod cycle_time

  if (cycle_position < half_cycle) then
    -- build the highlight
    set cycle_percent = float(cycle_position) / half_cycle
  else
    -- reduce the highlight
    set cycle_percent = 1 - float(cycle_position -
half_cycle) / half_cycle
  end if

  -- update the colors of highlighted shaders
  repeat with model_count = 1 to model_highlight_list.count
    set this_model =
custom_shader_list.getaProp(model_highlight_list.getPropAt(m
odel_count))

    if NOT(voidP(this_model)) then
      -- alter each shader
      repeat with shader_count = 1 to
this_model.shader.count
        -- get the shader props
        set shader_name =
this_model.shader.getPropAt(shader_count)
        set shader_diffuse =
this_model.shader[shader_count].diffuse
        set shader_ambient =
this_model.shader[shader_count].ambient

        -- set the shader to the interpolated values
        scene.shader(shader_name).diffuse =
interpolateColor(shader_diffuse, highlight_diffuse,
cycle_percent)
```

```
        scene.shader(shader_name).ambient =
interpolateColor(shader_ambient, highlight_ambient,
cycle_percent)

      end repeat
    end if

  end repeat

end


on highlightGroup group_name, event_name
  -- sets the models within the group to highlight and the
event name
  if (verifyGroup(group_name)) then
    repeat with group_child = 1 to
scene.group(group_name).child.count


highlightModel(scene.group(group_name).child[group_child].na
me, event_name)

    end repeat
  end if
end

on unHighlightGroup group_name
  -- removes the highlight from all models within the group
  if (verifyGroup(group_name)) then
    repeat with group_child = 1 to
scene.group(group_name).child.count


unHighlightModel(scene.group(group_name).child[group_child].
name)

    end repeat
  end if
end

on setModelOpacity model_name, opacity_percentage
  -- sets the opacity (relative) of a model
  -- if the model has not had custom shaders, then create
them

  -- check to see if the model exists
  if (verifyModel(model_name)) then

    -- see if this already has a shader turned on
    set this_model = custom_shader_list.getaProp(model_name)
    if voidP(this_model) then
      -- model currently doesn't have a shader attached to
it.
```

```
      -- create a new Shader
      set create_check = createCustomShader(model_name)
      if (create_check <> TRUE) then
        return 0    -- denotes model couldn't create a new
object
      end if

      this_model = custom_shader_list.getaProp(model_name)

      if voidP(this_model) then
        -- error creating the shader
        return 0    -- denotes model couldn't create a new
object
      end if

      model_opacity_list.addProp(model_name,
opacity_percentage)

    else if voidP(model_opacity_list.getaProp(model_name))
then
      -- is this in the opacity list?
      model_opacity_list.addProp(model_name,
opacity_percentage)
    else

      model_opacity_list[model_name] = opacity_percentage

    end if

    -- go through each shader for the model and apply the
current percentage blend
    -- on the stored value (partially transparent will
maintain ratio)

    repeat with shade_count = 1 to this_model.shader.count
      set this_shader =
this_model.shader.getPropAt(shade_count)
      set orig_blend = this_model.shader[shade_count].blend
      set final_blend = float(orig_blend *
opacity_percentage)

      scene.shader(this_shader).blend = final_blend
    end repeat

    return TRUE  -- denotes successful completion

  else
    return -1  -- denotes that model doesn't exist
  end if

end

on changeModelOpacity event_attributes, percentage
  --
  -- event attributes are:
  -- #model_name: <model name>
  -- #start_value: <blend percentage> OR #this
  -- #end_value: <blend percentage>

  set model_name = event_attributes.model_name
  set start_value = event_attributes.start_value
  set end_value = event_attributes.end_value

  if (verifyModel(model_name)) then

    if (start_value = #this) then
      if voidP(model_opacity_list.getaProp(model_name)) then
        -- model isn't in the opacity list - 100% on.
        start_value = 1.0
      else
        -- model is already in the opacity list
        start_value = model_opacity_list[model_name]
      end if
    end if

    if (start_value > end_value) then
      -- fade out
      set span = float(start_value) - end_value
      set opacity_percent = start_value - (span *
percentage)
    else if (start_value < end_value) then
      -- fade in
      set span = float(end_value) - start_value
      set opacity_percent = start_value + (span *
percentage)
    else
      -- they are equal, return TRUE
      return TRUE
    end if

    -- change the opacity
    setModelOpacity(model_name, opacity_percent)

    if (percentage = 1) then
      return TRUE
    else
      event_attributes.start_value = start_value
      return event_attributes
    end if

  else
    return -1   -- denotes that model doesn't exist
  end if


end
```

```
on changeGroupOpacity event_attributes, percentage
  --
  -- event attributes are:
  -- #group_name: <model name>
  -- #start_value: <blend percentage> OR #this
  -- #end_value: <blend percentage>

  if (verifyGroup(event_attributes.group_name)) then
    -- do this for every_model in the group
    repeat with model_count = 1 to
scene.group(event_attributes.group_name).child.count

      changeModelOpacity([#model_name:
scene.group(event_attributes.group_name).child[model_count].
name, #start_value: event_attributes.start_value,
#end_value: event_attributes.end_value], percentage)

    end repeat
  else
    return -1  -- denotes that group doesn't exist
  end if

end

on changeGroupColor event_attributes, percentage
  -- event attributes are:
  -- #group_name: <model name>
  -- #start_value: <blend percentage> OR #this
  -- #end_value: <blend percentage>

  if (verifyGroup(event_attributes.group_name)) then
    -- do this for every_model in the group
    repeat with model_count = 1 to
scene.group(event_attributes.group_name).child.count

      changeModelColor([#model_name:
scene.group(event_attributes.group_name).child[model_count].
name, #start_value: event_attributes.start_value,
#end_value: event_attributes.end_value], percentage)

    end repeat
  else
    return -1  -- denotes that group doesn't exist
  end if

end


on fadeOffModel model_name, duration
  -- makes this model fade out, then turns the model off.

  if (verifyModel(model_name)) then
    -- set the variables
    set model_opacity =
model_opacity_list.getaProp(model_name)

    if voidP(model_opacity) then
      model_opacity = 1
    end if

    set fadeout_name = "fadeout_" & the milliseconds & "_1_"
& model_name
    set turnoff_name = "fadeout_" & the milliseconds & "_2_"
& model_name
    set reset_name = "fadeout_" & the milliseconds & "_3_" &
model_name

    -- fadeout the event
    addEventToQueue([#name: fadeout_name, #command:
#change_opacity, #starttime: #now, #duration: duration,
#priority: #noverify, #attributes:[#model_name: model_name,
#start_value: #this, #end_value: 0], #hold_trigger: #none])

    -- set the turn off event
    addEventToQueue([#name: turnoff_name, #command:
#toggle_model, #starttime: #hold, #duration: 0, #priority:
#verify, #attributes:[#model_name: model_name, #toggle:
#off], #hold_trigger: fadeout_name])

    -- reset the opacity
    addEventToQueue([#name: reset_name, #command:
#change_opacity, #starttime: #hold, #duration: 0, #priority:
#verify, #attributes:[#model_name: model_name, #start_value:
#this, #end_value: model_opacity], #hold_trigger:
turnoff_name])

  else
    return -1  -- denotes that the model doesn't exist
  end if


end

on fadeOnModel model_name, duration
  --invisible_model_list.getaProp(model_name)
  if (verifyModel(model_name)) then
    -- set the variables
    set model_opacity =
model_opacity_list.getaProp(model_name)

    if voidP(model_opacity) then
      model_opacity = 1
    end if

    set reset_event = "fadein_" & the milliseconds & "_1_" &
model_name
```

```
    set turnon_event = "fadein_" & the milliseconds & "_2_"
& model_name
    set fadein_event = "fadein_" & the milliseconds & "_3_"
& model_name

    -- reset the opacity
    addEventToQueue([#name: reset_event, #command:
#change_opacity, #starttime: #now, #duration: 0, #priority:
#verify, #attributes:[#model_name: model_name, #start_value:
#this, #end_value: 0], #hold_trigger: #none])

    -- set the turn on event
    addEventToQueue([#name: turnon_event, #command:
#toggle_model, #starttime: #hold, #duration: 0, #priority:
#verify, #attributes:[#model_name: model_name, #toggle:
#on], #hold_trigger: reset_event])

    -- fadein the event
    addEventToQueue([#name: fadein_event, #command:
#change_opacity, #starttime: #hold, #duration: duration,
#priority: #noverify, #attributes:[#model_name: model_name,
#start_value: #this, #end_value: model_opacity],
#hold_trigger: turnon_event])

  end if

end


on changeModelColor event_attributes, percentage
  -- event attributes are:
  -- #model_name: <model name>
  -- #start_value: <shader_list> OR #this OR #original
  -- #end_value: <diffuse & ambient> OR #original


  set model_name = event_attributes.model_name



  -- is this a valid model?

  if (verifyModel(model_name)) then
    set this_model =
custom_shader_list.getaProp(event_attributes.model_name)

    -- is there a shader for this model?
    if voidP(this_model) then
      -- no, add a custom shader for this model
      set create_check = createCustomShader(model_name)
      if (create_check <> TRUE) then
        return 0    -- denotes model couldn't create a new
object
      end if
      this_model = custom_shader_list.getaProp(model_name)

      if voidP(this_model) then
        -- error creating the shader
        return 0    -- denotes model couldn't create a new
object
      end if

      model_color_list.addProp(model_name,
duplicate(this_model.shader))
    else
      -- is this in the MODEL COLOR LIST?
      set model_check =
model_color_list.getaProp(model_name)
      if voidP(model_check) then
        -- no, add it to the list
        model_color_list.addProp(model_name,
duplicate(this_model.shader))
      end if
    end if

    -- check the start value
    -- does this have an original setting?
    if (event_attributes.start_value = #original) then
      event_attributes.start_value =
model_color_list[model_name]
    else if (event_attributes.start_value = #this) then
      -- set shader values to current value
      event_attributes.start_value =
duplicate(this_model.shader)
    end if

    -- check the end value
    -- does this have an original setting?
    if (event_attributes.end_value = #original) then
      event_attributes.end_value =
model_color_list[model_name]
    end if

    -- repeat through each shader in this model list.
    repeat with shade_count = 1 to
event_attributes.start_value.count

      -- set the start values
      set start_list =
event_attributes.start_value[shade_count]
      set start_ambient = start_list.ambient
      set start_diffuse = start_list.diffuse

      -- set the end values
      if
voidP(event_attributes.end_value.getaProp(#ambient)) then
        -- this is a value list
        set end_list =
event_attributes.end_value[shade_count]
        set end_ambient = end_list.ambient
        set end_diffuse = end_list.diffuse
      else
        -- these are global to the model
        set end_ambient = event_attributes.end_value.ambient
        set end_diffuse = event_attributes.end_value.diffuse
      end if

      -- get the current color, based on percentage
      set current_ambient = interpolateColor(start_ambient,
end_ambient, percentage)
      set current_diffuse = interpolateColor(start_diffuse,
end_diffuse, percentage)

      -- apply the current colors
      set shader_name =
event_attributes.start_value.getPropAt(shade_count)
      scene.shader(shader_name).ambient = current_ambient
      scene.shader(shader_name).diffuse = current_diffuse

      -- update the custom shader list

custom_shader_list[model_name].shader[shader_name].ambient =
current_ambient

custom_shader_list[model_name].shader[shader_name].diffuse =
current_diffuse

    end repeat

    if (percentage = 1) then
      return TRUE
    else
      return event_attributes
    end if

  else
    return -1   -- denotes that this model does not exist
  end if

end
```

Catmull-Rom Spline Scripts

```
-- These scripts handle setup and determination of a point
along
-- a Catmull-Rom spline, defined by control points.
--
-- Catmull-Rom Spline Equation
-- P(u) = U(T)MB

global cr_M
global cr_MBx
global cr_MBy
global cr_MBz

on CR_MSetup
  -- sets up the M matrix for the Catmull-Rom Equation
  --             |-1  3 -3  1 |
  --             | 2 -5  4 -1 |
  -- M = 0.5 *   |-1  0  1  0 |
  --             | 0  2  0  0 |

  set catmull_matrix = [[-1,3,-3,1],[2,-5,4,-1],[-
1,0,1,0],[0,2,0,0]]

  cr_M = 0.5 * catmull_matrix

end


on CR_GlobalSegmentSetup prev_point, start_point, end_point,
next_point
  -- sets up the MB calculations for each (x,y,z) point
  -- builds the B(x), B(y), and B(z) matrix
  --
  --      | P(i-1) |  prev_point
  --      | P(i)   |  start_point
  -- B =  | P(i+1) |  end_point
  --      | P(i+2) |  next_point

  set Bx =
[[prev_point.x],[start_point.x],[end_point.x],[next_point.x]
]
  set By =
[[prev_point.y],[start_point.y],[end_point.y],[next_point.y]
]
  set Bz =
[[prev_point.z],[start_point.z],[end_point.z],[next_point.z]
]

  cr_MBx = multiplyMatrices(cr_M, Bx)
  cr_MBy = multiplyMatrices(cr_M, By)
  cr_MBz = multiplyMatrices(cr_M, Bz)
```

```
end

on CR_GetSegmentMB prev_point, start_point, end_point,
next_point
  -- returns the 3 MB matrices.  Used to store values when
globals will be
  -- changing, such as storing a spline motion, or multiple
simulataneous animations

  -- sets up the MB calculations for each (x,y,z) point
  -- builds the B(x), B(y), and B(z) matrix
  --
  --      | P(i-1) |  prev_point
  --      | P(i)   |  start_point
  -- B =  | P(i+1) |  end_point
  --      | P(i+2) |  next_point

  set Bx =
[[prev_point.x],[start_point.x],[end_point.x],[next_point.x]
]
  set By =
[[prev_point.y],[start_point.y],[end_point.y],[next_point.y]
]
  set Bz =
[[prev_point.z],[start_point.z],[end_point.z],[next_point.z]
]

  set MBx = multiplyMatrices(cr_M, Bx)
  set MBy = multiplyMatrices(cr_M, By)
  set MBz = multiplyMatrices(cr_M, Bz)

  return [#x: MBx, #y: MBy, #z: MBz]

end


on CR_GlobalGetPoint u
  -- determines a point along the current segment by
percentage of completion
  -- creates the U(transform) matrix, and multiplies by the
MB combination

  set Ut = [[(u * u * u), (u * u), (u), 1]]

  set Px = multiplyMatrices(Ut, cr_MBx)
  set Py = multiplyMatrices(Ut, cr_MBy)
  set Pz = multiplyMatrices(Ut, cr_MBz)

  return vector(Px[1][1], Py[1][1], Pz[1][1])

end

on CR_GetPoint u, MB
```

```
  -- determines a point along the current segment by
percentage of completion
  -- creates the U(transform) matrix, and multiplies by the
MB combination

  if listP(MB) then

    set Ut = [[(u * u * u), (u * u), (u), 1]]

    set Px = multiplyMatrices(Ut, MB.x)
    set Py = multiplyMatrices(Ut, MB.y)
    set Pz = multiplyMatrices(Ut, MB.z)

    return vector(Px[1][1], Py[1][1], Pz[1][1])
  else
    return -1 -- Bad MB Value
  end if

end

on CR_GetDerivative u, MB

  -- determines the derivative (directional vector) by
percentage of completion
  -- creates the U' matrix, and multiplies by the MB
combination

  if listP(MB) then
    set Ut = [[(3 * u * u), (2 * u), 1, 0]]

        set Px = multiplyMatrices(Ut, MB.x)
    set Py = multiplyMatrices(Ut, MB.y)
    set Pz = multiplyMatrices(Ut, MB.z)

    return vector(Px[1][1], Py[1][1], Pz[1][1])
  else
    return -1 -- Bad MB Value
  end if

end


on CR_GetSegmentDistanceData MB, point_number
  -- calculates the distances for the segment.
  -- returns list of distance - count is parameterization
  -- number of points used in calculation is determined by
"point_number"

  -- set up the initial point
  set cur_point = CR_GetPoint(0, MB)
  set distance_chart = []
  set total_dist = 0

  repeat with i = 1 to point_number
```

```
    set u = (1.0 / point_number) * i

    set new_point = CR_GetPoint(u, MB)
    set new_dist = cur_point.distanceTo(new_point)

    total_dist = total_dist + new_dist

    -- add to the chart
    distance_chart.append(total_dist)

    -- update the current point
    cur_point = new_point

  end repeat

  return distance_chart

end
```

Matrix Scripts

```
on newMatrix rows, columns
  -- creates a (ROWS x COLUMNS) list matrix
  -- populates each with 0
  -- returns the matrix

  if (rows > 0) and (columns > 0) then

    set newmatrix = []

    repeat with i = 1 to rows
      set newcolumn = []
      repeat with j = 1 to columns
        newcolumn.append(float(0))
      end repeat
      newmatrix.append(newcolumn)
    end repeat

    return newmatrix
  else
    return -1  -- denotes an invalid matrix parameter
  end if

end

on validateMatrix this_matrix
  -- Checks to see if this is a two dimensional, numeric
matrix
  -- repairs integers by converting to floats
  set matrix = this_matrix

  if (listP(matrix)) then
    if (listP(matrix[1])) then
      -- check structure
      set row_param = matrix.count
      set column_param = matrix[1].count

      if (row_param > 0) and (column_param > 0) then
        repeat with i = 1 to row_param
          if (matrix[i].count = column_param) then
            repeat with j = 1 to column_param
              -- if this is not a float, correct it
              if NOT(floatP(matrix[i][j])) then
                -- check to see if this is an integer
                if (integerP(matrix[i][j])) then
                  matrix[i][j] = float(matrix[i][j])
                else
                  return FALSE  -- THIS IS NOT A VALID
MATRIX
                end if
              end if
            end repeat
          else
            return FALSE      -- matrix is not valid
          end if
        end repeat

        -- the matrix is valid
        return matrix

      else
        return FALSE     -- matrix is not valid
      end if
    else
      return FALSE      -- matrix is not valid
    end if
  else
    return FALSE    -- matrix is not valid
  end if


end

on getMatrixParams this_matrix
  -- returns the row & column attributes for the matrix OR
  -- returns FALSE if there is a matrix problem
  -- this script assumes that "validateMatrix" has already
been run

  set matrix = this_matrix
  if listP(matrix) then
    set matrix_rows = matrix.count
    if (matrix_rows > 0) then
      if listP(matrix[1]) then
        set matrix_columns = matrix[1].count
        if (matrix_columns > 0) then

          -- return the dimensions of the matrix
          return [#rows: matrix_rows, #columns:
matrix_columns]

        end if
      else
        return FALSE -- not a matrix
      end if
    else
      return FALSE -- not a matrix
    end if
  else
    return FALSE   -- not a matrix
  end if

end
```

```
on multiplyMatrices this_matrixA, this_matrixB
  -- Multiplies two matrices (Matrix A x Matrix B)
  -- Returns the resultant matrix, OR
  --       0: Cannot Multiply Matrix (columnA / rowB match-up
error)
  --       -1: Invalid Matrix Error

  -- check matrix validity
  matrixA = validateMatrix(this_matrixA)
  matrixB = validateMatrix(this_matrixB)

  if (matrixA = FALSE) OR (matrixB = FALSE) then
    return -1   -- Invalid Matrix Error
  end if


  -- checks matrix dimensions
  matrixA_params = getMatrixParams(matrixA)
  matrixB_params = getMatrixParams(matrixB)

  if (matrixA_params = FALSE) OR (matrixB_params = FALSE)
then
    return -1   -- Invalid Matrix Error
  end if

  if (matrixA_params.columns = matrixB_params.rows) then
    -- set up the resultant matrix
    set result_matrix = newMatrix(matrixA_params.rows,
matrixB_params.columns)

    set v = matrixA_params.columns

    -- compute each result
    repeat with i = 1 to matrixA_params.rows
      repeat with j = 1 to matrixB_params.columns
        set this_entry = 0.0

        repeat with k = 1 to v
          this_entry = this_entry + (matrixA[i][k] *
matrixB[k][j])
        end repeat

        result_matrix[i][j] = this_entry

      end repeat
    end repeat
  else
    return 0   -- Cannot Multiply Matrices
  end if

  return result_matrix

end
```

```
on multiplyByScalar this_matrixA, this_scalar
  -- Multiplies a matrix by a scalar
  -- Returns the resultant matrix, OR
  --       0: Invalid Scalar Error
  --      -1: Invalid Matrix Error

  -- check matrix validity
  set matrixA = validateMatrix(this_matrixA)

  if (matrixA = FALSE) then
    return -1  -- Invalid Matrix Error
  end if

  matrixA_params = getMatrixParams(matrixA)

  if (matrixA_params = FALSE) then
    return -1  -- invalid matrix error
  end if

  -- check scalar validity
  set scalar = this_scalar

  if (floatP(scalar) = FALSE) AND (integerP(scalar) = FALSE)
then
    return 0  -- invalid scalar error
  end if

  set result_matrix = newMatrix(matrixA_params.rows,
matrixA_params.columns)

  -- multiply each value by scalar
  repeat with i = 1 to matrixA_params.rows
    repeat with j = 1 to matrixA_params.columns
      result_matrix[i][j] = matrixA[i][j] * scalar
    end repeat
  end repeat

  -- return the resultant matrix
  return result_matrix

end
```

Normalized Distance-Time Functions

```
-- Normalized Distance / Time Functions
-- Written 3/12/03 by Tom Corbett (tcorbett@vt.edu)
--
-- These functions calculate a maximum velocity,
-- distance by time, and time by distance.

-- Distance and Time are normalized, that is, their values
-- are from 0 to 1.  This can be thought of as all values
-- are a decimal percentage of the total distance and time.
-- Total Distance/Time should be stored elsewhere.


on findNormalVo t1, t2
  -- returns the normalized maximum velocity of the function
  -- takes two arguments, t1 and t2.
  -- t1: (acceleration time / total time)
  -- t2: ((total time - deceleration time) / total time)

  set vo = (2.0 / (t2 - t1 + 1.0))

  return vo

end


on findDistanceByTime time_percent, accel_percent,
decel_percent
  -- returns the normalized distance at time_t (current
time)
  -- takes three arguments, time_t, time_t1, time_t2
  -- time_percent:  (time variable / total time)
  -- accel_percent: (acceleration time / total time)
  -- decel_percent: ((total time - deceleration time) /
total time)

  -- set the variables
  set timeT = time_percent
  set t1 = accel_percent
  set t2 = (1.0 - decel_percent)

  -- get the max velocity
  set vo = findNormalVo(t1, t2)

  -- the acceleration leg
  if (timeT > 0.0) AND (timeT < t1) then
    set distance_t = vo * ((timeT * timeT) / (2.0 * t1))
    return distance_t
  end if

  -- the straight run leg
  if (timeT >= t1) AND (timeT <= t2) then
    set distance_t = vo * ((t1 / 2.0) + (timeT - t1))
    return distance_t
  end if

  -- the deceleration leg
  if (timeT > t2) AND (timeT < 1) then
    set distance_t = (vo * ((t1 / 2.0) + (t2 - t1))) + ((vo
- ((vo * ((timeT - t2)/(1.0 - t2))) / 2.0)) * (timeT - t2))
    return distance_t
  end if

  if (timeT = 0) then
    return 0
  end if

  if (timeT = 1) then
    return 1
  end if

  return -1  -- there was an error calculating the
percentage

end


on findTimeByDistance distance_percent, accel_percent,
decel_percent
  -- returns the normalized time at distance_percent
  -- takes three arguments, time_t, time_t1, time_t2
  -- distance_percent:  (distance variable / total distance)
  -- accel_percent: (acceleration time / total time)
  -- decel_percent: ((total time - deceleration time) /
total time)

  -- set the variables
  set distT = distance_percent
  set t1 = accel_percent
  set t2 = 1.0 - decel_percent

  -- get the max velocity
  set vo = findNormalVo(t1, t2)

  -- set the area variables
  set d1 = (0.5 * (t1 * vo))
  set d2 = (vo * (t2 - t1))
  set d3 = (0.5 * ((1.0 - t2) * vo))


  -- the acceleration leg
  if (distT > 0.0) AND (distT < d1) then
    set time_percent = sqrt((distT / vo) * (2.0 * t1))
    return time_percent
  end if
  -- the straight run
  if (distT >= d1) AND (distT <= (d1 + d2)) then
    set time_percent = (distT / vo) + (t1 / 2.0)
    return time_percent
  end if

  -- the deceleration run
  if (distT > (d1 + d2)) AND (distT < 1.0) then
    set time_percent = (1.0 - ((1 - t2) * sqrt((1.0 - distT)
/ d3)))
    return time_percent
  end if

  -- take care of the 0 / 1 values
  if (distT = 0) then
    return 0
  else if (distT = 1) then
    return 1
  end if

  return -1  -- there was an error calculating the
percentage


end
```

Motion Setup Scripts

```
on MotionSetup  object_name, object_type, start_time,
duration, start_pos, motion_sequence, accel_time, decel_time
  -- this system parses motions / motion series into
individual motion
  -- events that are added to the event queue.
  --
  -- Takes the following arguments:
  -- object_name: <name> or #camera.  The name of the object
to be moved.
  -- object_type: #model, #group, or #light.  (#none for
camera)
  -- start_time: <milliseconds> or #now.
  -- duration: <milliseconds>.  The duration for the entire
segment / series.
  -- start_pos: <point> or #here.  The start point for the
model.
  --                                #here is only valid for
#now motions.
  -- motion_sequence: <motion list>  The list of motions
that are to be executed
  --                                  as part of this
series.
  -- accel_time: <milliseconds> or #none.  The time from 0
to maximum velocity.  (Ease-In time)
  -- decel_time: <milliseconds> or #none.  The time from
maximum velocity to 0.  (Ease-Out time)
  --
  -- Motions parsing steps are:
  --    - Error Checking
  --    - Calculate the parameter / distance tables for each
segment (5% approximation)
  --    - Calculate overall distance
  --    - Determine each segment start-time / duration by
distance
  --    - Set timechart for each segment (5% approximation)
  --    - Build each segment event.  Send to event queue


  -- Overall Event Variables
  set is_camera = FALSE
  set event_name_base = "Motion_" & object_name & "_" & the
milliseconds



  -- ERROR CHECKING

  -- check the model / group / light

  case (object_type) of
```

```
    #model:
      if NOT(verifyModel(object_name)) then
        return -1  -- model name not verified
      end if
    #group:
      if NOT(verifyGroup(object_name)) then
        return -1  -- group name not verified
      end if
    #light:
      if NOT(verifyLight(object_name)) then
        return -1  -- light name not verified
      end if
    #camera:
      is_camera = TRUE
    otherwise
      return -1  -- object_type is not valid
  end case


  -- start_time check
  if (start_time <> #now) then
    if NOT(integerP(start_time)) or (start_time < 0) then
return -1  -- start_time error
  end if

  -- duration check
  if NOT(integerP(duration)) or (duration < 0) then return -
1  -- duration error

  -- accel / decel checks
  if (accel_time = #none) then accel_time = 0
  if (decel_time = #none) then decel_time = 0

  if NOT(integerP(accel_time)) or (accel_time < 0) then
return -1  -- accel error
  if NOT(integerP(decel_time)) or (decel_time < 0) then
return -1  -- decel error

  if ((accel_time + decel_time) > duration) then return -1
-- accel/decel/duration mismatch

  -- start position check
  if (start_pos = #here) then
    case (object_type) of
      #model:
        start_pos = getModelPosition(object_name)
      #group:
        start_pos = getGroupPosition(object_name)
      #light:
        start_pos = getLightPosition(object_name)
      #camera:
        start_pos = getCameraPosition()
    end case
  end if
  if NOT(ilk(start_pos) = #vector) then return -1  -- not a
valid vector

  -- motion series check
  if listP(motion_sequence) then

    -- check each motion in the sequence to ensure it is
properly formatted
    repeat with seg_count = 1 to motion_sequence.count
      set this_seg = motion_sequence[seg_count]

      case (this_seg.type) of
        #linear_absolute:
          if NOT(ilk(this_seg.data) = #vector) then return -
1  -- not a valid point
        #linear_relative:
          if NOT(ilk(this_seg.data) = #vector) then return -
1  -- not a valid point
        #orbit_absolute:
          if listP(this_seg.data) then
            set temp_angle = this_seg.data.getaProp(#angle)
            set temp_pivot = this_seg.data.getaProp(#pivot)
            set temp_axis = this_seg.data.getaProp(#axis)
            set temp_rotate =
this_seg.data.getaProp(#rotate)

            if voidP(temp_angle) then
              return -1  -- bad data set
            else
              if NOT(integerP(temp_angle) OR
floatP(temp_angle)) then
                return -1  -- bad angle data
              end if
            end if

            if voidP(temp_pivot) then
              return -1  -- bad data set
            else
              if NOT(ilk(temp_pivot) = #vector) then return
-1  -- not a vector
            end if

            if voidP(temp_axis) then
              return -1  -- bad data set
            else
              if NOT(ilk(temp_axis) = #vector) then
                return -1  -- not a vector
              else if (temp_axis.length = 0) then
                return -1   -- not a valid axis vector
              end if

            end if
            if voidP(temp_rotate) then
              temp_rotate = FALSE
            end if


          else
            return -1 --  invalid data for orbit
          end if
        #spline_path:
          if listP(this_seg.data) then

            -- check each entry in the data for valid
points.
            repeat with this_point = 1 to
this_seg.data.count
              if NOT(ilk(this_seg.data[this_point]) =
#vector) then return -1  -- bad vector data
            end repeat
          else
            return -1  -- bad spline data
          end if

        otherwise
          return -1  -- bad segment type
      end case

    end repeat
  else
    return -1  -- motion sequence must be a list
  end if

  -- BUILD EACH SEGMENT, CALCULATE THE DISTANCE / DISTANCE
TABLES
  set segment_list = []
  set current_pos = start_pos
  set current_dist = 0.0
  set segment_count = 1

  repeat with seg_index = 1 to motion_sequence.count
    set this_segment = motion_sequence[seg_index]
    set segment_data = this_segment.data

    case (this_segment.type) of
      #linear_absolute:
        -- get the distance
        set segment_length = getLinearDistance(current_pos,
segment_data)

        if (segment_length > 0.0) then
          -- build the segment item, and add to the SEGMENT
LIST
          set segment_item = [#id: segment_count, #type:
#linear, #start_pos: current_pos, #end_pos: segment_data,
#distance: segment_length, #speedchart: VOID]
```

```
          segment_list.append(segment_item)
        end if

        -- update the distance, position, and segment count
data
        current_dist = current_dist + segment_length
        current_pos = segment_data
        segment_count = segment_count + 1

      #linear_relative:
        -- get the distance
        set segment_length =
getLinearDistance(vector(0,0,0), segment_data)

        if (segment_length > 0.0) then
          -- build the segment item, and add to the SEGMENT
LIST
          set segment_item = [#id: segment_count, #type:
#linear, #start_pos: current_pos, #end_pos: (current_pos +
segment_data), #distance: segment_length, #speedchart: VOID]
          segment_list.append(segment_item)
        end if

        -- update the distance, position, and segment count
data
        current_dist = current_dist + segment_length
        current_pos = current_pos + segment_data
        segment_count = segment_count + 1

      #orbit_absolute:
        -- get the distance
        set segment_length = getArcDistance(current_pos,
segment_data.angle, segment_data.pivot, segment_data.axis)

        if (segment_length > 0.0) then
          -- build the segment item, and add it to the
SEGMENT LIST
          set segment_item = [#id: segment_count, #type:
#orbit, #start_pos: current_pos, #angle: segment_data.angle,
#pivot: segment_data.pivot, #axis: segment_data.axis,
#rotate: segment_data.rotate, #distance: segment_length,
#speedchart: VOID]
          segment_list.append(segment_item)
        end if

        -- update the distance, position, and segment count
data
        current_dist = current_dist + segment_length
        current_pos = getArcPoint(current_pos,
segment_data.angle, segment_data.pivot, segment_data.axis)
        segment_count = segment_count + 1

      #spline_path:
        -- break the spline into individual segments by
points.
        set spline_count = segment_data.count
        repeat with spline = 1 to spline_count

          -- set the points for the previous point /
start_point
          if (spline = 1) then
            -- if this is not the first in a motion
sequence, smooth the start to linear
            if (seg_index > 1) then
              set previous_point = current_pos -
(segment_data[spline] - current_pos)
            else
              set previous_point = current_pos
            end if
            set start_point = current_pos
          else
            previous_point = start_point
            start_point = current_pos
          end if

          -- set the points for the end_point / next_point
          if (spline = spline_count) then
            -- this is the last end-point
            set end_point = segment_data[spline]
            set next_point = end_point
          else
            set end_point = segment_data[spline]
            set next_point = segment_data[spline + 1]
          end if

          -- build the MB matrix
          set segment_MB = CR_GetSegmentMB(previous_point,
start_point, end_point, next_point)

          -- calculate the distance
          set segment_distance_chart =
CR_GetSegmentDistanceData(segment_MB, 20)  -- 5%
approximation
          set segment_length =
segment_distance_chart[segment_distance_chart.count]

          if (segment_length > 0.0) then
            -- divide by zero test, don't include segment if
it has no length

            set param_distance_chart =
segment_distance_chart / float(segment_length)

            -- build the segment item and add it to the
SEGMENT LIST
```

```
            set segment_item = [#id: segment_count, #type:
#spline, #MB_matrix: segment_MB, #distance: segment_length,
#distchart: param_distance_chart, #speedchart: VOID]
            segment_list.append(segment_item)
          end if


          -- update the current position, distance, and
segment count data
          current_pos = end_point
          current_dist = current_dist + segment_length
          segment_count = segment_count + 1

        end repeat

    end case

  end repeat

  set overall_distance = current_dist


  -- Accel / Decel Determination
  set accel_percent = float(accel_time) / duration
  set decel_percent = float(decel_time) / duration

  -- get the accel / decel distance values
  set accel_end = findDistanceByTime(accel_percent,
accel_percent, decel_percent) * overall_distance
  set decel_start = (1.0 - findDistanceByTime(decel_percent,
accel_percent, decel_percent)) * overall_distance

  set segment_start_time = 0.0
  set current_dist = 0

  repeat with segment_index = 1 to segment_list.count

    set this_segment = segment_list[segment_index]

    -- Timing Determinination

    -- get the parameterized start/end distance of this
segment
    set segment_start_dist = float(current_dist) /
overall_distance
    set segment_end_dist = float(current_dist +
this_segment.distance) / overall_distance

    -- get the parameterized end time, duration of this
segment
    set segment_end_time =
findTimeByDistance(segment_end_dist, accel_percent,
decel_percent)
```

```
    set segment_duration = segment_end_time -
segment_start_time


    if (current_dist < accel_end) OR ((current_dist +
this_segment.distance) > decel_start) then
      -- build for an acceleration velocity or deceleration
velocity


      -- set up speed-chart (5% approximation)
      set time_chart = []
      set time_count = 20

      set time_increment = segment_duration / time_count

      repeat with time_index = 1 to time_count
        set time_value = segment_start_time +
(time_increment * time_index)
        set real_distance = (findDistanceByTime(time_value,
accel_percent, decel_percent) * overall_distance)
        set param_distance = (real_distance -
(segment_start_dist * overall_distance)) /
this_segment.distance
        time_chart.append(param_distance)
      end repeat

      this_segment.speedchart = time_chart

    else

      -- test this with the spline function
      this_segment.speedchart = #constant

      --      -- build a constant velocity
      --      case this_segment.type of
      --        #linear:
      --          this_segment.speedchart = #constant
      --        #orbit:
      --          this_segment.speedchart = #constant
      --        #spline:
      --          -- set up spline speed-chart  (5%
approximation)
      --          set time_chart = []
      --          set time_count = 20
      --
      --          repeat with time_index = 1 to time_count
      --            -- parameterized segment time
      --            set time_percent = (1.0 / time_count) *
time_index
      --            time_chart.append(time_percent *
this_segment.distance)
      --          end repeat
      --
      --          this_segment.speedchart = time_chart
      --
      --      end case
    end if

    -- Set the Event Variables
    set m_eventName = event_name_base & "_" &
this_segment.id
    set m_holdTrigger = #none


    case this_segment.type of
      #linear:
        set m_eventType = #linear_motion
      #orbit:
        set m_eventType = #orbit_motion
      #spline:
        set m_eventType = #spline_motion
    end case

    if (segment_index = 1) then
      if (start_time = #now) then
        set m_startTime = #now
      else
        set m_startTime = start_time
      end if
    else
      if (start_time = #now) then
        set m_startTime = #hold
        set m_holdTrigger = event_name_base & "_" &
segment_list[segment_index - 1].id
      else
        set m_startTime = integer(segment_start_time *
duration) + start_time
      end if
    end if

    set m_durationTime = integer(segment_duration *
duration)
    set m_priority = #noverify

    -- set segment type attributes
    case (this_segment.type) of
      #linear:
        set s_attributes = [#object_name: object_name,
#object_type: object_type, #linear_type: #absolute,
#start_pos: this_segment.start_pos, #end_pos:
this_segment.end_pos, #speedchart: this_segment.speedchart,
#completed: 0, #target: #path]
      #orbit:
        set s_attributes = [#object_name: object_name,
#object_type: object_type, #orbit_type: #absolute,
#start_pos: this_segment.start_pos, #angle:
this_segment.angle, #pivot: this_segment.pivot, #axis:
```

```
this_segment.axis, #rotate: this_segment.rotate,
#speedchart: this_segment.speedchart, #completed: 0,
#target: #path]
      #spline:
        set s_attributes = [#object_name: object_name,
#object_type: object_type, #MB_matrix:
this_segment.MB_matrix,  #distchart: this_segment.distchart,
#speedchart: this_segment.speedchart, #target: #path]

    end case

    -- create the event
    set motion_queue_event = [#name: m_eventName, #command:
m_eventType, #starttime: m_startTime, #duration:
m_durationTime, #priority: m_priority, #attributes:
s_attributes, #hold_trigger: m_holdTrigger]

    -- add the event to the queue
    addEventToQueue(motion_queue_event)

    -- update the distance / time variables
    current_dist = current_dist + this_segment.distance
    segment_start_time = segment_end_time

  end repeat

end


on getLinearDistance position_a, position_b
  -- returns the distance from
  return position_a.distanceTo(position_b)
end

on getLinearPoint position_a, position_b, percentage
  -- returns a percentage of the distance
  return ((position_b - position_a) * percentage) +
position_a
end



--on getArcDistance start_point, arc_angle, arc_pivot,
arc_axis
--  -- calculates the distance of a rotation by calculating
the radius,
--  -- then returning the percentage of a complete rotation
times the circumference
--
--  if (arc_axis.length = 0) then
--    return -1  -- axis is not valid
--  end if
--
```

```
--  -- check to see if pivot and start_point share the
axis...
--
--
--  set arc_radius = ((start_point -
arc_pivot).cross(arc_axis).length) / arc_axis.length
--
--  set angle_percent = arc_angle / 360.0
--
--  -- return rotation * pi * radius-squared
--  return angle_percent * (PI * (arc_radius * arc_radius))
--
--end

on getArcDistance start_point, arc_angle, arc_pivot,
arc_axis
  -- calculates the distance of a rotation by calculating
the radius
  -- then returning the percentage of a complete rotation
times the circumference

  if (arc_axis.length = 0) then
    return -1  -- axis is not valid
  end if

  -- get the angle between the items
  set hyp_vector = start_point - arc_pivot

  set test_angle = hyp_vector.angleBetween(arc_axis)

  -- check for special cases
  if (test_angle = 90) then
    -- this is perpendicular
    set angle_percent = arc_angle / 360.0
    set arc_radius = hyp_vector.length
    return angle_percent * 2.0 * PI * arc_radius
  else if (test_angle = 0) then
    -- this is colinear
    return 0.0
  else
    set arc_radius = (hyp_vector.cross(arc_axis).length) /
arc_axis.length
    set angle_percent = arc_angle / 360.0
    return angle_percent * 2.0 * PI * arc_radius
  end if

end



on getArcPoint start_point, arc_angle, arc_pivot, arc_axis
  -- returns a point along a circular curve
  set temp_transform = transform()
  temp_transform.position = start_point
  temp_transform.rotate(arc_pivot, arc_axis, arc_angle)

  return temp_transform.position
end

on getTangentVector start_point, arc_angle, arc_pivot,
arc_axis
  -- returns the normalized tangent direction vector by
rotating the radius about the center point
  set start_location = getArcPoint(start_point, arc_angle,
arc_pivot, arc_axis)

  set tangent_vector = getArcPoint(start_location, 90,
arc_pivot, arc_axis) - arc_pivot
  tangent_vector = getNormalized(tangent_vector)

  if (arc_angle < 0.0) then
    -- if the arc angle is negative, resultant direction
vector is positive
    return tangent_vector
  else
    -- if the arc angle is positive, resultant direction
vector is negative
    return -1 * tangent_vector
  end if

end


--on getSpeedChartParametric speed_chart, time_percent
--  -- compares values in a speed chart and returns the
parametric completion value
--  -- speed chart lists normalized distance over time
(mainly for accel / decel purposes)
--
--  -- chart index determined by integer test
--
--  -- distance determined by:
--  --
--  -- dist[i] + (((time_percent - time[i]) / (time[i+1] -
time[i])) * (dist[i+1] - dist[i]))
--
--  set chart_inc = 1.0 / speed_chart.count
--  set chart_location = (time_percent /  chart_inc)
--  set chart_int = integer(chart_location)
--
--  set round_check = chart_location - chart_int
--
--  if (round_check > 0) then
--    -- rounded down
--    if (chart_int = 0) then
--      -- i value is zero, i + 1 is first item
--       return ((time_percent / chart_inc) * speed_chart[1])
--     else
--       set i = chart_int
--       set time0 = chart_inc * chart_int
--       set time1 = chart_inc * (chart_int + 1)
--
--       set dist_val = speed_chart[i] + (((time_percent -
time0) / (time1 - time0)) * (speed_chart[i + 1] -
speed_chart[i]))
--
--       return dist_val
--
--     end if
--
--
--   else if (round_check < 0) then
--     -- round up
--     if (chart_int = 1) then
--       -- i value is zero, i + 1 is first item
--       return ((time_percent / chart_inc) * speed_chart[1])
--     else
--       set i = chart_int - 1
--       set time0 = chart_inc * (chart_int - 1)
--       set time1 = chart_inc * chart_int
--
--       set dist_val = speed_chart[i] + (((time_percent -
time0) / (time1 - time0)) * (speed_chart[i + 1] -
speed_chart[i]))
--
--       return dist_val
--
--     end if
--
--   else
--     if (chart_int = 0) then
--       return 0.0
--     else
--       -- percentage = parametered time
--       return speed_chart[chart_int]
--
--     end if
--
--   end if
--
--end

on getSpeedChartParametric speed_chart, time_percent
  -- compares values in a speed-chart chart and returns the
parametric u value
  -- speed chart lists u-value over increments of time
(accel / decel applications)

  -- u value determined by:
  --
  -- u[i] + ((time% - [i])/([i+1] - [i])) * (u[i+1]-u[i])


  set time_value = 0.0
  set time_inc = 1.0 / speed_chart.count

  repeat with chart_index = 1 to speed_chart.count

    time_value = chart_index * time_inc

    if (time_value > time_percent) then
      -- falls in the previous span
      chart_index = chart_index - 1

      if (chart_index > 0) then
        set last_time = chart_index * time_inc
        set return_value = (speed_chart[chart_index]) +
((time_percent - last_time) / (time_inc)) *
(speed_chart[chart_index + 1] - speed_chart[chart_index])

        return return_value


      else
        -- return the zero to 1st increment value
        return (time_percent / time_inc) * speed_chart[1]
      end if


    else if (time_value = time_percent) then
      -- this is the value
      return speed_chart[chart_index]
    end if

  end repeat

end


on getDistanceChartParametric dist_chart, dist_percent
  -- compares values in a distance chart and returns the
parametric u value
  -- distance chart lists normalized distance over
increments of "u" (percentage along a spline)

  -- u value determined by:
  --
  -- [i] + ((dist% - dist[i])/(dist[i+1] - dist[i])) *
([i+1]-[i])


  set distance_value = 0.0
  set u_inc = 1.0 / dist_chart.count
  repeat with chart_index = 1 to dist_chart.count

    distance_value = dist_chart[chart_index]

    if (dist_percent < distance_value) then
      -- falls in the previous span
      chart_index = chart_index - 1

      if (chart_index > 0) then
        set last_distance = dist_chart[chart_index]
        set return_value = (u_inc * chart_index) +
((dist_percent - last_distance) / (distance_value -
last_distance)) * (u_inc)

        return return_value

      else
        -- return the zero to 1st increment value
        return (dist_percent / dist_chart[1]) * (u_inc)
      end if


    else if (dist_percent = distance_value) then
      -- this is the value
      return (u_inc * chart_index)
    end if

  end repeat

end


on rotationSetup event_name, object_name, object_type,
start_time, duration, rotation_atts, accel_time, decel_time,
event_hold, return_event
  -- this system parses the rotation into a call to the
event queue
  --
  -- Takes the following arguments:
  -- object_name: <name>.  The name of the object to be
moved.
  -- object_type: (OBJECT TYPE IS DETERMINED BY VERIFY
OBJECT COMMAND)
  -- start_time: <milliseconds> or #now.
  -- duration: <milliseconds>.  The duration for the entire
segment / series.
  -- rotation_atts: <attribute list>.  Sets the attributes
for this rotation.
  -- accel_time: <milliseconds> or #none.  The time from 0
to maximum velocity.  (Ease-In time)
  -- decel_time: <milliseconds> or #none.  The time from
maximum velocity to 0.  (Ease-Out time)
  -- event_hold: <event name> or #none.  The event that this
holds for.
```

```
  -- return_event: TRUE or FALSE.  If TRUE, returns the
event name for holds.
  --
  -- Rotation Attributes:
  -- rotate_type:  #self.  Rotates around the axis as
defined by object coordinate system
  --               #world. Rotates around the axis as
defined by world coordinate system
  --               #orientation.  Rotates the point at
orientation of the object.
  --               #camera_angle:  Rotates camera to
relative <pan angle> and <tilt angle>
  --               #camera_direction:  Rotates camera along
axis angle
  --               #camera_orientation:  Rotates camera
orientation (PAN ANGLE ONLY)
  -- rotate_axis:  <axis vector>
  -- rotate_angle:  <angle>
  --
  -- Rotation Steps:
  --  1.  Error Checking
  --  2.  Parameterization (Ease In / Out)
  --  3.  Event Building
  --  4.  Send to Queue

  -- Overall Event Variables
  set object_type = verifyObject(object_name)
  -- set event_name = "Rotate_" & object_name & "_" & the
milliseconds

  -- 1.  ERROR CHECKING
  if voidP(object_type) then
    return -1  --- object does not exist
  end if

  -- start_time check
  if (start_time <> #now) then
    if NOT(integerP(start_time)) or (start_time < 0) then
return -1  -- start_time error
  end if

  -- duration check
  if NOT(integerP(duration)) or (duration < 0) then return -
1  -- duration error

  -- accel / decel checks
  if (accel_time = #none) then accel_time = 0
  if (decel_time = #none) then decel_time = 0

  if NOT(integerP(accel_time)) or (accel_time < 0) then
return -1  -- accel error
  if NOT(integerP(decel_time)) or (decel_time < 0) then
return -1  -- decel error
```

```
  if ((accel_time + decel_time) > duration) then return -1
-- accel/decel/duration mismatch

  set rotation_type = rotation_atts.type
  -- appropriate object check
  if (object_type = #camera) then
    -- only camera events are valid
    case (rotation_type) of
      #camera_angle:
        set pan_angle = rotation_atts.getaProp(#pan)
        set tilt_angle = rotation_atts.getaProp(#tilt)

        if (voidP(pan_angle)) then return -1  -- bad
attributes
        if (voidP(tilt_angle)) then return -1  -- bad
attributes

        if NOT(integerP(pan_angle) OR floatP(pan_angle))
then return -1  -- bad angle
        if NOT(integerP(tilt_angle) OR floatP(tilt_angle))
then return -1  -- bad angle

      #camera_direction:
        set axis_vec = rotation_atts.getaProp(#axis)
        set angle_val = rotation_atts.getaProp(#angle)

        if voidP(axis_vec) then
          return -1 -- bad attributes
        else if NOT(vectorP(axis_vec)) then
          return -1 -- bad attributes
        else if (axis_vec.length = 0) then
          return -1 -- bad vector
        end if

        if voidP(angle_val) then
          return -1 -- bad attributes
        else if NOT(integerP(angle_val) OR
floatP(angle_val)) then
          return -1 -- bad angle value
        end if

      #camera_orientation:
        set pan_angle = rotation_atts.getaProp(#pan)
        if voidP(pan_angle) then
          return -1  -- bad angle value
        else if NOT(integerP(pan_angle) OR
floatP(pan_angle)) then
          return -1  -- bad angle value
        end if
      otherwise
        return -1  -- not a valid type
    end case
  else
    -- other object type
    set axis_vec = rotation_atts.getaProp(#axis)
    set angle_val = rotation_atts.getaProp(#angle)

    if voidP(axis_vec) then
      return -1 -- bad attributes
    else if NOT(vectorP(axis_vec)) then
      return -1 -- bad attributes
    else if (axis_vec.length = 0) then
      return -1 -- bad vector
    end if

    if voidP(angle_val) then
      return -1 -- bad attributes
    else if NOT(integerP(angle_val) OR floatP(angle_val))
then
      return -1 -- bad angle value
    end if
  end if

  -- 2. PARAMETERIZATION
  -- get variables for parameterized accel / decel values
  if (duration <> 0) then
    set accel_percent = float(accel_time) / duration
    set decel_percent = float(decel_time) / duration
  else
    set accel_percent = 0
    set decel_percent = 0
  end if


  -- (will evaluate the parameterized completion at run-
time)


  -- 3.  BUILD THE EVENT
  case (rotation_type) of
    #camera_angle:
      -- set the camera rotation attributes
      set event_type = #rotate_camera
      set event_attributes = [#rotate_type: #pantilt, #pan:
pan_angle, #tilt: tilt_angle, #orientation: FALSE,
#accel_time: accel_percent, #decel_time: decel_percent,
#completed: 0]
    #camera_direction:
      -- set the camera rotation (axis / angle) attributes
      set event_type = #rotate_camera
      set event_attributes = [#rotate_type: #axisangle,
#axis: axis_vec, #angle: angle_val, #orientation: FALSE,
#accel_time: accel_percent, #decel_time: decel_percent,
#completed: 0]
    #camera_orientation:
      -- set the camera orientation attributes
      set event_type = #rotate_camera
      set event_attributes = [#rotate_type: #pantilt, #pan:
pan_angle, #tilt: 0, #orientation: TRUE, #accel_time:
accel_percent, #decel_time: decel_percent, #completed: 0]
    #orientation:
      -- set the object orientation attributes
      set event_type = #rotate_object
      set event_attributes = [#object_name: object_name,
#object_type: object_type, #rotation_type: #orientation,
#axis: axis_vec, #angle: angle_val,  #accel_time:
accel_percent, #decel_time: decel_percent, #completed: 0]
    #self:
      set event_type = #rotate_object
      set event_attributes = [#object_name: object_name,
#object_type: object_type, #rotation_type: #self, #axis:
axis_vec, #angle: angle_val, #accel_time: accel_percent,
#decel_time: decel_percent, #completed: 0]
    #world:
      -- set the object rotation events
      set event_type = #rotate_object
      set event_attributes = [#object_name: object_name,
#object_type: object_type, #rotation_type: #world, #axis:
axis_vec, #angle: angle_val, #accel_time: accel_percent,
#decel_time: decel_percent, #completed: 0]
  end case



  set rotation_queue_event = [#name: event_name, #command:
event_type, #starttime: start_time, #duration: duration,
#priority: #noverify, #attributes: event_attributes,
#hold_trigger: event_hold]



  -- 4.  ADD THE EVENT TO THE QUEUE
  addEventToQueue(rotation_queue_event)

  if (return_event) then
    return event_name
  end if



end




on targetTransitionSetup object_name, start_time, duration,
target_atts, accel_time, decel_time
  -- this system parses the rotation into a call to the
event queue
  --
  -- Takes the following arguments:
```

```
  -- object_name: <name>.  The name of the object to be
moved.
  -- object_type: (OBJECT TYPE IS DETERMINED BY VERIFY
OBJECT COMMAND)
  -- start_time: <milliseconds> or #now.
  -- duration: <milliseconds>.  The duration for the entire
segment / series.
  -- rotation_atts: <attribute list>.  Sets the attributes
for this rotation.
  -- accel_time: <milliseconds> or #none.  The time from 0
to maximum velocity.  (Ease-In time)
  -- decel_time: <milliseconds> or #none.  The time from
maximum velocity to 0.  (Ease-Out time)
  --
  -- Target Attributes:
  --   #type:        | #data
  --     #position   |   <position vector>
  --     #object     |   <object name>
  --     #direction  |   <direction vector>
  --     #path       |   NO DATA (path is evaluated & stored
in object target list
  --
  -- ALSO...
  -- #persistent: TRUE or FALSE.  If false, completion of
targeting simply runs.
  --                             If true, target is added
into target list upon completion
  --
  -- Target Transition Steps:
  --  1.  Error Checking
  --  2.  Parameterization (Ease In / Out)
  --  3.  Event Building
  --  4.  Send to Queue


  -- Overall Event Variables
  set object_type = verifyObject(object_name)
  set event_name = "Target_" & object_name & "_" & the
milliseconds

  -- 1.  ERROR CHECKING
  if voidP(object_type) then
    return -1  --- object does not exist
  end if

  -- start_time check
  if (start_time <> #now) then
    if NOT(integerP(start_time)) or (start_time < 0) then
return -1  -- start_time error
  end if

  -- duration check
  if NOT(integerP(duration)) or (duration < 0) then return -
1  -- duration error
```

```
  -- accel / decel checks
  if (accel_time = #none) then accel_time = 0
  if (decel_time = #none) then decel_time = 0

  if NOT(integerP(accel_time)) or (accel_time < 0) then
return -1  -- accel error
  if NOT(integerP(decel_time)) or (decel_time < 0) then
return -1  -- decel error

  if ((accel_time + decel_time) > duration) then return -1
-- accel/decel/duration mismatch

  -- attribute testing
  set target_type = target_atts.getaProp(#type)
  set target_data = target_atts.getaProp(#data)
  set target_persists = target_atts.getaProp(#persistent)

  case (target_type) of
    #position:
      if NOT(vectorP(target_data)) then return -1  -- data
is not a vector
    #object:
      if voidP(verifyObject(target_data)) then return -1  --
not a valid object
    #direction:
      if NOT(vectorP(target_data)) then return -1  -- data
is not a vector
    #path:
      -- nothing
    #none:
      -- nothing
    otherwise
      return -1  -- bad attributes / type value
  end case

  -- 2. PARAMETERIZATION
  -- get variables for parameterized accel / decel values
  if (duration <> 0) then
    set accel_percent = float(accel_time) / duration
    set decel_percent = float(decel_time) / duration
  else
    set accel_percent = 0
    set decel_percent = 0
  end if

  -- (will evaluate the parameterized completion at run-
time)


  -- 3.  BUILD THE EVENT

  if (target_type = #none) then
```

```
    -- this turns off the targetting for this object.  it is
an instant change.
    set duration = 0
    set accel_percent = 0
    set decel_percent = 0

    set event_attributes = [#object_name: object_name,
#object_type: object_type, #start_dir: #this, #target_type:
#none, #target_data: #none, #persistent: FALSE, #accel_time:
accel_percent, #decel_time: decel_percent, #completed: 0]

  else
    set event_attributes = [#object_name: object_name,
#object_type: object_type, #start_dir: #this, #target_type:
target_type, #target_data: target_data, #persistent:
target_persists, #accel_time: accel_percent, #decel_time:
decel_percent, #completed: 0]
  end if

  set transition_queue_event = [#name: event_name, #command:
#target_transition, #starttime: start_time, #duration:
duration, #priority: #noverify, #attributes:
event_attributes, #hold_trigger: #none]

  -- 4.  ADD EVENT TO THE QUEUE
  addEventToQueue(transition_queue_event)

end
```

Motion Interpolation Scripts

```
global scene
global cameraobject

on interpolateLinearMotion event_attributes,
percent_complete
  -- takes a linear motion call from the event queue,  and
  -- interpolates it based on a percentage complete value

  set object_type = event_attributes.object_type
  set object_name = event_attributes.object_name

  -- error-check type
  case object_type of
    #camera:
      set valid_object = TRUE
    #model:
      set valid_object = verifyModel(object_name)
    #group:
      set valid_object = verifyGroup(object_name)
    #light:
      set valid_object = verifyLight(object_name)

  end case

  if (valid_object) then
    -- check the start value
    if (event_attributes.start_pos = #here) then
      case object_type of
        #camera:
          event_attributes.start_pos = getCameraPosition()
        #model:
          event_attributes.start_pos =
getModelPosition(object_name)
        #group:
          event_attributes.start_pos =
getGroupPosition(object_name)
        #light:
          event_attributes.start_pos =
getLightPosition(object_name)

      end case
    end if

    -- get the correct percentage
    if (event_attributes.speedchart = #constant) then
      u_value = percent_complete
    else
      u_value =
getSpeedChartParametric(event_attributes.speedchart,
percent_complete)
    end if
    -- get the point to move to
    if (event_attributes.linear_type = #absolute) then
      set current_point =
getLinearPoint(event_attributes.start_pos,
event_attributes.end_pos, u_value)
      -- update the object position

      case object_type of
        #camera:
          setCameraPosition(current_point)
        #model:
          setModelPosition(object_name, current_point)
        #group:
          setGroupPosition(object_name, current_point)
        #light:
          setLightPosition(object_name, current_point)
      end case


      if (event_attributes.target = #path) then
        -- set the direction for this object
        set directional_vec = event_attributes.end_pos -
event_attributes.start_pos
        set exec_test =
setDirectionalTargetPath(object_name, directional_vec)
        -- if NOT(exec_test = -1) then
event_attributes.target = #set
      end if



    else if (event_attributes.linear_type = #relative) then
      set current_point = getLinearPoint(vector(0,0,0),
event_attributes.end_pos, u_value)

      set translate_vec = current_point -
event_attributes.completed

      case object_type of
        #camera:
          translateCameraWorld(translate_vec)
        #model:
          scene.model(object_Name).translate(translate_vec,
#world)
        #group:
          scene.group(object_name).translate(translate_vec,
#world)
        #light:
          scene.light(object_name).translate(translate_vec,
#world)
      end case

      event_attributes.completed = current_point
    end if

    if (percent_complete = 1) then
      return TRUE
    else
      return event_attributes
    end if

  else
    return -1  -- this is not a valid object, cancel the
call
  end if

end

on interpolateOrbitMotion event_attributes, percent_complete
  -- takes a orbit motion call from the event queue,  and
  -- interpolates it based on a percentage complete value

  set object_type = event_attributes.object_type
  set object_name = event_attributes.object_name

  -- error-check type
  case object_type of
    #camera:
      set valid_object = TRUE
    #model:
      set valid_object = verifyModel(object_name)
    #group:
      set valid_object = verifyGroup(object_name)
    #light:
      set valid_object = verifyLight(object_name)
  end case

  if (valid_object) then
    -- check the start value
    if (event_attributes.start_pos = #here) then
      case object_type of
        #camera:
          event_attributes.start_pos = getCameraPosition()
        #model:
          event_attributes.start_pos =
getModelPosition(object_name)
        #group:
          event_attributes.start_pos =
getGroupPosition(object_name)
        #light:
          event_attributes.start_pos =
getLightPosition(object_name)
      end case
    end if

    -- get the correct percentage
    if (event_attributes.speedchart = #constant) then
      u_value = percent_complete
    else
      u_value =
getSpeedChartParametric(event_attributes.speedchart,
percent_complete)
    end if

    -- get the point to move to
    if (event_attributes.orbit_type = #absolute) then
      set current_angle = event_attributes.angle * u_value
      set current_point =
getArcPoint(event_attributes.start_pos, current_angle,
event_attributes.pivot, event_attributes.axis)
      -- update the object position



      case (event_attributes.rotate) of
        FALSE:
          -- only move the object around the point
          case (object_type) of
            #camera:
              setCameraPosition(current_point)
            #model:
              setModelPosition(object_name, current_point)
            #group:
              setGroupPosition(object_name, current_point)
            #light:
              setLightPosition(object_name, current_point)
          end case
        TRUE:
          set this_percent = percent_complete -
event_attributes.completed
          set inc_angle = event_attributes.angle *
this_percent
          -- move and rotate this object around the point.
          case (object_type) of
            #camera:
              setCameraPosition(current_point)
              rotateCameraByAngle(event_attributes.axis,
inc_angle)
            #model:
              setModelPosition(object_name, current_point)
              setModelWorldRotation(object_name,
event_attributes.axis, inc_angle)
            #group:
              setGroupPosition(object_name, current_point)
              setGroupWorldRotation(object_name,
event_attributes.axis, inc_angle)
            #light:
              setLightPosition(object_name, current_point)
              setLightWorldRotation(object_name,
event_attributes.axis, inc_angle)
```

```
          end case
          event_attributes.completed = u_value
      end case

      if (event_attributes.target = #path) then
        --        setDirectionalTarget(object_name, #path,
vector(-1,0,0))
        event_attributes.target = #update
      end if

      if (event_attributes.target = #update) then
        -- update the target with the latest information
        set directional_vec =
getTangentVector(event_attributes.start_pos, current_angle,
event_attributes.pivot, event_attributes.axis)
        setDirectionalTargetPath(object_name, -
directional_vec)
      end if

    else if (event_attributes.orbit_type = #relative) then
      set current_angle = event_attributes.angle * u_value
      set current_point = getArcPoint(vector(0,0,0),
current_angle, event_attributes.pivot,
event_attributes.axis)

      set translate_vec = current_point -
event_attributes.completed

      case object_type of
        #camera:
          translateCameraWorld(translate_vec)
        #model:
          scene.model(object_Name).translate(translate_vec,
#world)
        #group:
          scene.group(object_name).translate(translate_vec,
#world)
        #light:
          scene.light(object_name).translate(translate_vec,
#world)
      end case

      event_attributes.completed = current_point

    end if

    if (percent_complete = 1) then
      return TRUE
    else
      return event_attributes
    end if

  else
```

```
    return -1  -- this is not a valid object, cancel the
call
  end if

end

on interpolateSplineMotion event_attributes,
percent_complete
  -- takes a spline motion call from the event queue,  and
  -- interpolates it based on a percentage complete value

  set object_type = event_attributes.object_type
  set object_name = event_attributes.object_name

  -- error-check type
  case object_type of
    #camera:
      set valid_object = TRUE
    #model:
      set valid_object = verifyModel(object_name)
    #group:
      set valid_object = verifyGroup(object_name)
    #light:
      set valid_object = verifyLight(object_name)
  end case

  if (valid_object) then

    -- no need to check start value - all values contained
in segment MB matrix

    -- get the correct percentage
    if (event_attributes.speedchart = #constant) then
      u_value =
getDistanceChartParametric(event_attributes.distchart,
percent_complete)
    else
      u_value =
getDistanceChartParametric(event_attributes.distchart,
getSpeedChartParametric(event_attributes.speedchart,
percent_complete))
    end if

    -- get the point to move to  (all splines are absolute
value)

    set current_point = CR_GetPoint(u_value,
event_attributes.MB_matrix)
    -- update the object position

    case object_type of
      #camera:
        setCameraPosition(current_point)
      #model:
        setModelPosition(object_name, current_point)
      #group:
        setGroupPosition(object_name, current_point)
      #light:
        setLightPosition(object_name, current_point)
    end case

    if (event_attributes.target = #path) then
      --      setDirectionalTarget(object_name, #path,
vector(-1,0,0))
      event_attributes.target = #update
    end if

    if (event_attributes.target = #update) then
      -- update the target with the latest information
      set directional_vec = CR_getDerivative(u_value,
event_attributes.MB_matrix)
      directional_vec = getNormalized(directional_vec)
      setDirectionalTargetPath(object_name, directional_vec)
    end if

    if (percent_complete = 1) then
      return TRUE
    else
      return event_attributes
    end if

  else
    return -1  -- this is not a valid object, cancel the
call
  end if

end

on interpolateDirectionalRotation start_dir, end_dir,
percent_complete
  -- rotates a direction to an interpolative value from a
start to end direction
  --
  -- calculates the axis of rotation through cross product

  set direction1 = getNormalized(start_dir)
  set direction2 = getNormalized(end_dir)

  set axis_vector = direction1.cross(direction2)
  set angle_val = direction1.angleBetween(direction2)

  -- correct for co-linear directions - up is the rotational
direction
  if (axis_vector.length = 0) then
    axis_vector = vector(0,1,0)
  end if
  set int_direction = rotateDirectionAboutAxis(direction1,
axis_vector, (angle_val * percent_complete))

  return int_direction

end

on interpolateObjectRotation event_attributes, time_complete
  -- rotates an object about an axis and angle, and resets
the orientation if necessary
  --
  -- Expected Attributes:
  -- #object_name: <name>
  -- #object_type: #model, #group, OR #light
  -- #rotation_type: #self, #world, OR #orientation
  -- #axis: <axis vec>.  Axis to rotate about.
  -- #angle: <angle>.  Angle to rotate about axis
  -- #accel_time: <time percentage>
  -- #decel_time: <time percentage>
  -- #completed: <percent previously completed>
  if voidP(verifyObject(event_attributes.object_name)) then
    return -1  -- object doesn't exist
  else


    -- determine parameterized completion
    set percent_complete = findDistanceByTime(time_complete,
event_attributes.accel_time, event_attributes.decel_time)

    -- calculate the angle to be used
    set this_percent = percent_complete -
event_attributes.completed

    set object_name = event_attributes.object_name
    set axis_vec = event_attributes.axis
    set angle_val = (event_attributes.angle * this_percent)

    -- rotate based on type
    case (event_attributes.object_type) of
      #model:
        case (event_attributes.rotation_type) of
          #self:
            setModelSelfRotation(object_name, axis_vec,
angle_val)
          #world:
            setModelWorldRotation(object_name, axis_vec,
angle_val)
          #orientation:
            rotateModelOrientation(object_name, axis_vec,
angle_val)
        end case
      #group:
        case (event_attributes.rotation_type) of
          #self:
            setGroupSelfRotation(object_name, axis_vec,
angle_val)
          #world:
            setGroupWorldRotation(object_name, axis_vec,
angle_val)
          #orientation:
            rotateGroupOrientation(object_name, axis_vec,
angle_val)
        end case
      #light:
        case (event_attributes.rotation_type) of
          #self:
            setLightSelfRotation(object_name, axis_vec,
angle_val)
          #world:
            setLightWorldRotation(object_name, axis_vec,
angle_val)
          #orientation:
            rotateLightOrientation(object_name, axis_vec,
angle_val)
        end case
    end case

    -- update the complete time
    event_attributes.completed = percent_complete

    if (time_complete = 1) then
      return TRUE
    else
      return event_attributes
    end if


  end if

end

on interpolateCameraRotation event_attributes, time_complete
  -- pans / tilts the camera, or rotates direction about an
axis
  --
  -- Expected Attributes:
  -- #rotate_type: #pantilt
  --                     #pan: <angle>
  --                     #tilt: <angle>
  --               #axisangle
  --                     #axis: <axis vector>
  --                     #angle: <angle>
  -- #orientation: TRUE or FALSE.  If true, then reset the
pan angle orientation
  -- #accel_time: <time percentage>
  -- #decel_time: <time percentage>
  -- #completed: <percent previously completed>
  -- determine parameterized completion
  set percent_complete = findDistanceByTime(time_complete,
event_attributes.accel_time, event_attributes.decel_time)

  -- calculate the angle to be used
  set this_percent = percent_complete -
event_attributes.completed

  if (event_attributes.rotate_type = #pantilt) then
    if (event_attributes.orientation) then
      -- this is an orientation event
      set pan_angle = event_attributes.pan * this_percent
      rotateCameraFromDirection(pan_angle)
    else
      -- percentage pan & tilt the camera
      set pan_angle = event_attributes.pan * this_percent
      set tilt_angle = event_attributes.tilt * this_percent
      panCameraRelative(pan_angle)
      tiltCameraRelative(tilt_angle)
    end if
  else if (event_attributes.rotate_type = #axisangle) then
    -- get the current direction, rotate, then point at new
direction
    set this_direction = getCameraDirection()
    set this_axis = event_attributes.axis
    set this_angle = event_attributes.angle * this_percent
    set new_direction =
rotateDirectionAboutAxis(this_direction, this_axis,
this_angle)
    pointCameraAtDirection(new_direction)
  end if

  -- update the complete time
  event_attributes.completed = percent_complete

  if (time_complete = 1) then
    return TRUE
  else
    return event_attributes
  end if


end

on interpolateTargetTransition event_attributes,
time_complete
  -- rotates the object as a transition between targets.
Sets persistent targets
  --
  -- Expected Attributes:
  --   #object_name: <name>
  --   #object_type: #model, #group, #light, or #camera
  --   #start_dir: #this   Start direction is determined on
first run
```

```
  --   #target_type: #none, #position, #direction, #object,
#path
  --   #target_data: <type dependant data>
  --   #persistent: TRUE or FALSE    If true, the target is
to the target list.
  --   #completed: <percentage completed>

  if voidP(verifyObject(event_attributes.object_name)) then
    return -1  -- object doesn't exist
  else

    set object_name = event_attributes.object_name
    set target_type = event_attributes.target_type
    set target_data = event_attributes.target_data



    -- check for the first run
    if (event_attributes.start_dir = #this) then
      -- if the orientation has been altered, reset it.
      -- resetObjectOrientation(object_name)

      if (target_type = #path) then
        -- if this is a path object, set the transition data
catch
        setDirectionalTarget(object_name, #transition,
getObjectDirection(object_name))
      else
        -- delete any current targets that may be in the
object target list
        deleteDirectionalTarget(object_name)
      end if

      -- set the initial direction
      event_attributes.start_dir =
getObjectDirection(object_name)

    end if

    -- evaluate the end direction
    case (target_type) of
      #none:
        -- the target has been removed from the list
        return TRUE
      #position:
        set end_direction = target_data -
getObjectPosition(object_name)
      #direction:
        set end_direction = target_data
      #object:
        set end_direction = getObjectPosition(target_data) -
getObjectPosition(object_name)
      #path:
        set end_direction =
getDirectionTargetData(object_name)
    end case


    -- determine parameterized completion
    set percent_complete = findDistanceByTime(time_complete,
event_attributes.accel_time, event_attributes.decel_time)

    -- calculate the angle to be used
    -- set this_percent = percent_complete -
event_attributes.completed

    -- calculate the current direction
    set current_direction =
interpolateDirectionalRotation(event_attributes.start_dir,
end_direction, percent_complete)

    -- face the object to the current direction
    pointObjectAtDirection(object_name, current_direction)

    -- update the complete time
    event_attributes.completed = percent_complete

    if (time_complete = 1) then
      -- if this is a persistent event, add it to the queue
      if (target_type <> #none) then
        if (event_attributes.persistent) then
          setDirectionalTarget(object_name, target_type,
target_data)
        end if

      end if
      return TRUE
    else
      return event_attributes
    end if


  end if

end
```

Directional Targetting Scripts

```
global scene
global object_target_list

on setDirectionalTarget object_name, target_type,
target_data
  -- adds an object to the target list
  -- checks that the object actually exists

  set object_type = verifyObject(object_name)

  if voidP(object_type) then
    return -1  -- denotes that this object doesn't exist
  end if

  set target_props =
object_target_list.getaProp(object_name)

  if voidP(target_props) then
    set target_props = [#object_type: object_type,
#target:[#type: target_type, #data: target_data]]
    object_target_list.addProp(object_name, target_props)
  else
    object_target_list[object_name].target = [#type:
target_type, #data: target_data]
  end if

end

on deleteDirectionalTarget object_name
  -- if this exists in the object targetting list, deletes
that instance
  set this_object = object_target_list.getaProp(object_name)

  if listP(this_object) then
    object_target_list.deleteProp(object_name)
  end if
end

on setDirectionalTargetPath object_name, directional_vec
  -- if this exists in the object list, and it is set to
path, sets the target to the directional vector
  set target_check =
object_target_list.getaProp(object_name)
  if listP(target_check) then
    -- this does exist in the list
    set target_type =
object_target_list[object_name].target.type
    if (target_type = #path) OR (target_type = #transition)
then
      object_target_list[object_name].target.data =
directional_vec
```

```
    else
      return -1 -- denotes this is not there
    end if
  else
    return -1  -- denotes this is not there
  end if
end

on getDirectionTargetType object_name
  -- if this exists in the object list, returns the target
type
  set target_check =
object_target_list.getaProp(object_name)
  if listP(target_check) then
    return target_check.target.type
  else
    return #none
  end if

end

on getDirectionTargetData object_name
  -- if this exists in the object list, returns the data
  set target_check =
object_target_list.getaProp(object_name)
  if listP(target_check) then
    return target_check.target.data
  else
    return vector(0,0,0)
  end if

end



on updateObjectTargetting
  -- runs through the target list, and points all objects in
the list where
  -- they should be posted

  repeat with target_index = 1 to object_target_list.count
    -- initialize the variable to prevent overlap
    set pointtype = #none
    set pointloc = #none

    set object_name =
object_target_list.getPropAt(target_index)
    set object_type =
object_target_list[target_index].object_type
    set target_atts =
object_target_list[target_index].target


    case (target_atts.type) of
```

```
      #position:
        set pointtype = #position
        set pointloc = target_atts.data
      #object:
        set pointtype = #position
        set pointloc = getObjectPosition(target_atts.data)
        if (pointloc = -1) then
          set pointtype = #error
        end if
      #direction:
        set pointtype = #direction
        set pointloc = target_atts.data
      #path:
        set pointtype = #direction
        set pointloc = target_atts.data
    end case

    if (pointtype = #position) then
      pointObjectAtPosition(object_name, pointloc)
    else if (pointtype = #direction) then
      pointObjectAtPathDirection(object_name, pointloc)
    end if

  end repeat

end


on rotateDirectionAboutAxis object_direction, axis_vector,
angle
  -- rotates a direction about an axis, and returns the
resultant direction
  set this_transform = transform()

  this_transform.position = object_direction
  this_transform.rotate(vector(0,0,0), axis_vector, angle)

  return this_transform.position

end
```

Object Scripts

```
global scene

on verifyObject object_name
  -- checks model, group, and light list for this object
  -- returns the type or void if not found.

  if (verifyModel(object_name)) then
    return #model
  end if

  if (verifyGroup(object_name)) then
    return #group
  end if

  if (verifyLight(object_name)) then
    return #light
  end if

  if (object_name = "camera") then
    return #camera
  end if

end

on getObjectPosition object_name
  -- finds the type of an object, and returns the
corresponding position
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      return getModelPosition(object_name)
    #group:
      return getGroupPosition(object_name)
    #light:
      return getLightPosition(object_name)
    #camera:
      return getCameraPosition()
    otherwise
      return -1  -- denotes an error
  end case

end


on getObjectDirection object_name
  -- finds the type of an object, and returns the
corresponding direction
  set this_type = verifyObject(object_name)

  case (this_type) of
```

```
    #model:
      return getModelDirection(object_name)
    #group:
      return getGroupDirection(object_name)
    #light:
      return getLightDirection(object_name)
    #camera:
      return getCameraDirection()
    otherwise
      return -1  -- denotes an error
  end case
end


on getObjectSelfTransform object_name
  -- finds the type of object, then returns it's self
transform
  set this_type = verifyObject(object_name)
  case (this_type) of
    #model:
      return duplicate(scene.model(object_name).transform)
    #group:
      return duplicate(scene.group(object_name).transform)
    #light:
      return duplicate(scene.group(object_name).transform)
    #camera:
      return 0  -- no transform data for the object
    otherwise
      return -1  -- denotes an error
  end case
end

on setObjectSelfTransform object_name, this_transform
  -- finds the type of object, then returns it's self
transform
  set this_type = verifyObject(object_name)
  case (this_type) of
    #model:
      scene.model(object_name).transform = this_transform
    #group:
      scene.group(object_name).transform = this_transform
    #light:
      scene.group(object_name).transform = this_transform
    #camera:
      return 0  -- no transform data for the object
    otherwise
      return -1  -- denotes an error
  end case
end


on pointObjectAtPosition object_name, point_pos
  -- finds the object type, then points it at this position
  set this_type = verifyObject(object_name)
  case (this_type) of
    #model:
      pointModelAtPosition(object_name, point_pos)
    #group:
      pointGroupAtPosition(object_name, point_pos)
    #light:
      pointLightAtPosition(object_name, point_pos)
    #camera:
      pointCameraAtPosition(point_pos)
    otherwise
      return -1   -- denotes an error
  end case

end

on pointObjectAtDirection object_name, direction_vec
  -- finds the object type, then points it at this position
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      pointModelAtDirection(object_name, direction_vec)
    #group:
      pointGroupAtDirection(object_name, direction_vec)
    #light:
      pointLightAtDirection(object_name, direction_vec)
    #camera:
      pointCameraAtDirection(direction_vec)
    otherwise
      return -1   -- denotes an error
  end case

end

on pointObjectAtPathDirection object_name, direction_vec
  -- finds the object type, then points it at this position
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      pointModelAtDirection(object_name, direction_vec)
    #group:
      pointGroupAtDirection(object_name, direction_vec)
    #light:
      pointLightAtDirection(object_name, direction_vec)
    #camera:
      pointCameraAtPathDirection(direction_vec)
    otherwise
      return -1   -- denotes an error
  end case

end
```

```
on executeResetObjectOrientation event_attributes
  return
resetObjectOrientation(event_attributes.object_name)
end


on resetObjectOrientation object_name
  -- finds the object type, then resets the direction vector
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      resetModelOrientation(object_name)
    #group:
      resetGroupOrientation(object_name)
    #light:
      resetLightOrientation(object_name)
    #camera:
      resetCameraOrientation()
    otherwise
      return -1   -- denotes an error
  end case
  return TRUE

end



on highlightObject object_name
  -- finds the object type, then highlights it
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      highlightModel(object_name)
    #group:
      highlightGroup(object_name)

  end case
end

on unhighlightObject object_name
  -- finds the object type, then removes the highlight
  set this_type = verifyObject(object_name)

  case (this_type) of
    #model:
      unHighlightModel(object_name)
    #group:
      unHighlightGroup(object_name)
  end case
end
```

File Handling Scripts

```
on getFileTitle filePath

  set thisFileName = getFileName(filePath)
  idswitch(".")
  set thisFileTitle = thisFileName.item[1]
  idend
  return thisFileTitle

end

on checkNameValid thisString
  -- validates a filename string by checking for invalid
characters

  if voidP(thisString) or (thisString = "") then
    return FALSE
  else

    set invalid = ["\", "/", ":", "&", "*", "?", QUOTE, "<",
">", "|", SPACE, TAB, BACKSPACE, ENTER, RETURN]
    set c = 1

    repeat while (c <= thisString.char.count)
      if (getOne(invalid, thisString.char[c]) <> 0) then
        return FALSE
      else
        c = c + 1
      end if
    end repeat

    return TRUE

  end if

end


on checkStringValid thisString
  -- checks for invalid characters in a filepath string

  if voidP(thisString) or (thisString = "") then
    return FALSE
  else

    set invalid = [QUOTE, "|", BACKSPACE, ENTER, RETURN]
    set c = 1

    repeat while (c <= thisString.char.count)
      if (getOne(invalid, thisString.char[c]) <> 0) then
        return FALSE
      else
        c = c + 1
      end if
    end repeat

    return TRUE

  end if
end


on checkFileValid filePath
  -- tests for the existance of a file
  -- if the filepath is valid, returns TRUE
  -- if filepath is invalid, returns FALSE

  set fileTest = new(xtra "fileIO")
  fileTest.openFile(filePath, 1)
  if (status(fileTest) = 0) then
    fileTest.closeFile()
    fileTest = 0
    return TRUE
  else
    fileTest.closeFile()
    fileTest = 0
    return FALSE
  end if


end




on checkHexColor colorVal
  -- validates the hexadecimal format for a color
  -- Input: <hexcolor>
  -- Output:
  --      <hexcolor> (if hexcolor is valid)
  --      FALSE (if hexcolor is invalid)

  set valid = TRUE

  set validchar =
["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e"
,"f","A","B","C","D","E","F"]

  set validcount = the number of chars in colorVal

  if (validcount = 7) then
    if (colorVal.char[1] = "#") then
      set i = 2
      repeat while (i < validcount)
        if(validchar.getOne(colorVal.char[i]) = 0) then
```

```
          valid = FALSE
        end if
        i = i + 1
      end repeat
    else
      valid = FALSE
    end if
  else if (validcount = 6) then
    set i = 1
    repeat while (i < validcount)
      if(validchar.getOne(colorVal.char[i]) = 0) then
        valid = FALSE
      end if
      i = i + 1
    end repeat
    if(valid) then
      colorVal = "#" & colorVal
    end if
  else
    valid = FALSE
  end if

  if(valid) then
    return colorVal
  else
    return FALSE
  end if

end

on checkExtension fileString, ext
  set valid = FALSE

  idswitch(".")
  set itemCheck = the number of items in fileString


  if(itemCheck = 1) then
    fileString = fileString & "." & ext
    valid = TRUE
  else if(itemCheck = 2) then
    if(fileString.item[2] = ext) then
      valid = TRUE
    else
      put ext into fileString.item[2]
      valid = TRUE
    end if
  else
    valid = FALSE
  end if

  idend

  if(valid) then
```

```
    return fileString
  else
    return FALSE
  end if

end



on getFilePath fileString

  set fName = ""
  idswitch("\")
  set fCount = the number of items in fileString
  set i = 1
  repeat while (i < fCount)
    fName = fName & fileString.item[i]  & "/"
    i = i + 1
  end repeat
  idend

  if (fName <> "") then
    return fName
  else
    return 0
  end if

end



on getRPDir fileString

  set fName = ""
  idswitch("\")
  set fCount = the number of items in fileString
  set i = 1
  repeat while (i < fCount)
    if(i = 1) then
      fName = fileString.item[i]
    else
      fName = fName & "/" & fileString.item[i]
    end if
    i = i + 1
  end repeat
  idend

  if (fName <> "") then
    return fName
  else
    return 0
  end if
```

```
end

on getFileName fileString

  set fName = ""

  idswitch("\")
  fName = the last item in fileString
  idend
  if(fName <> "") then

    -- double check for backslash errors
    idswitch("/")
    if (the number of items in fName > 1) then
      fName = the last item in fileString
    end if
    idend

    if (fName <> "") then
      return fName
    else
      return 0
    end if

  else
    return 0
  end if
end

on getFileExt file_name

  idswitch(".")

  set itemCheck = the number of items in file_name

  if (itemCheck > 1) then
    return file_name.item[itemCheck]
  else
    return 0
  end if

  idend()
end


on getLastDir fileString

  set fName = ""

  idswitch("/")
  fName = the last item in fileString
  fName2 = fileString.item[((the number of items in
fileString) - 1)]
  idend
  if(fName <> "") then
    return fName
  else

    return fName2
  end if


end





on CR2CRLF source
  -- this command adds the correct linefeed character
  -- for a windows txt file.

  set dest to ""
  repeat while true
    set theOffset to offset(numToChar(13), source)
    if NOT theOffset then exit repeat
    put char 1 to theOffset of source after dest
    put numToChar(10) after dest
    delete char 1 to theOffset of source
  end repeat
  put source after dest
  return dest
end CR2CRLF

on CRLF2CR source
  -- removes the windows txt linefeed character

  set dest to ""
  repeat while true
    set theOffset to offset(numToChar(10), source)
    if NOT theOffset then exit repeat
    put char 1 to (theOffset - 1) of source after dest
    delete char 1 to theOffset of source
  end repeat
  put source after dest
  return dest
end CRLF2CR

on idswitch idchar
  global idbase

  idbase = the itemDelimiter
  set the itemDelimiter = idchar

end

on idend
  global idbase
  set the itemDelimiter = idbase
end




on findMediaObjects mediaTypeList, mediaAtt
  -- argument: [type(s)...], #attribute
  -- Searches through all existing objects, returns list of
hits
  -- each hit is a list of ID number and attribute
  -- returns 0 if none found
  -- returns -1 if error
  --
  -- addition: #all will return attribute for each object

  global allObjects

  if listP(mediaTypeList) then
    set types = duplicate(mediaTypeList)
  else
    return -1
  end if

  set returnlist = []

  set allIndex = 1
  repeat while (allIndex <= allObjects.count)
    set tempindex = duplicate(allObjects[allIndex])
    if (types.getOne(#all)) then
      typecheck = 1
    else
      set typecheck = types.getOne(tempindex.type)
    end if
    if (typecheck <> 0) then
      set thisatt = tempindex.getaProp(mediaAtt)
      if voidP(thisatt) then
        return -1
      else
        returnlist.append([#ID: tempindex.getaProp(#ID),
#att:thisatt])
      end if
    end if
    allIndex = allIndex + 1
  end repeat

  if (returnlist = []) then
    return 0
  else
    return returnlist
  end if

end
```

```
on getItemID thisItem, thisList
  -- argument: item, [[#id:num, #att:string or num]...]
  -- searches through a media list and returns a list of
ID's where
  -- the item matches the att.
  -- returns just the item if only one is found
  -- returns 0 if none found
  -- returns -1 if error

  if listP(thisList) then
    set returnList = []
  else
    return -1
  end if

  set lc = 1
  repeat while lc <= thisList.count
    if listP(thisList[lc]) then
      if (thisList[lc].att = thisItem) then
        returnList.append(thisList[lc].ID)
      end if
    else
      return -1
    end if
    lc = lc + 1
  end repeat

  if (returnList.count = 0) then
    return 0
  else if (returnList.count = 1) then
    return returnList[1]
  else
    return returnList
  end if

end

on getItemValid itemID, thisList
  -- argument: itemID, [[#id:num, #att:string or num]...]
  -- searches through a media list and returns whether or
not
  -- a list item is found
  -- returns TRUE if itemID is found
  -- returns 0 if itemID is not found


  if NOT(listP(thisList)) then
    return FALSE
  end if

  set lc = 1
  repeat while lc <= thisList.count
    if listP(thisList[lc]) then
```

```
      if (thisList[lc].id = itemID) then
        return TRUE
      end if
    else
      return FALSE
    end if
    lc = lc + 1
  end repeat

  return FALSE

end

on getItemAtt itemID, thisList
  -- argument: itemID, [[#id:num, #att:string or num]...]
  -- searches through the list and returns the attribute of
the
  -- first matching itemID

  if NOT(listP(thisList)) then
    return -1
  end if

  set lc = 1
  repeat while lc <= thisList.count
    if listP(thisList[lc]) then
      if (thisList[lc].id = itemID) then
        return thisList[lc].att
      end if
    else
      return -1
    end if
    lc = lc + 1
  end repeat

  return FALSE

end
```

Proximity Detection Scripts

```
global proximity_object_list
global active_proximity

on proximityListSetup
  proximity_object_list = [:]
  active_proximity = []
end

on addProximityObject obj_name, obj_position, obj_type,
obj_props, obj_trigger, obj_active
  -- Creates a proximity shape register within the proximity
object list.
```

```
  -- if the object is active, adds the list to the
active_proximity list
  -- PROPERTIES:
  -- obj_name:  <string> - the unique name of this object
  -- obj_position: <position list> - [#x, #y, #z]
  --     #x: <xval> - x value of position
  --     #y: <yval> - y value of position
  --     #z: <zval> - z value of position
  -- obj_type:  #box, #cylinder, or #sphere  - the type of
the object
  -- obj_props:  <object size properties - vary on type>
  --      type.box:  [#w:<width>, #l:<length>, #h:<height>]
  --      type.cylinder:  [#r: <radius>, #h:<height>]
  --      type.sphere: [#r:<radius>]
  -- obj_trigger:  <trigger name> - the event triggered by
proximity
  -- obj_active: TRUE or FALSE - is the object active?

  -- verify the object name does not already exist
  if (verifyProximityObject(obj_name)) then
    return FALSE -- object already exists
  end if

  -- verify the position object




end

on verifyProximityObject obj_name
  if (proximity_object_list.getOne(obj_name)) then
    return TRUE
  else
    return FALSE
  end if
end
```

General Scripts

```
-- GENERAL SCRIPTS
--
-- General sorting & ID scripts (that have no other catagory
to call home.)


on getNamedList full_list, list_name
  -- returns the complete list properties by matching a
nested #name attribute
  set list_idx = getNestedListByValue(full_list, #name,
list_name)
  if voidP(list_idx) then
    return VOID
  else
    return full_list[list_idx]
  end if

end

on getNamedListIndex full_list, list_name
  -- checks "full_list" for nested lists with #name
attribute
  return getNestedListByValue(full_list, #name, list_name)
end

on getNestedListByValue top_list, list_att, att_value
  -- retrieves a nested list by testing for a matching
attribute.

  set li = 1
  repeat while (li <= top_list.count)
    set l_idx = top_list[li].getaProp(list_att)
    if (l_idx = att_value) then
      return li
    end if
    li = li + 1
  end repeat

  -- nothing was found
  return VOID

end

on sendStatus status_text
  member("status_indicator").text = status_text
  updateStage
end

on getCameraStats

  set camvec = getCameraPosition()
  set camdir = getCameraDirection()
  set campan = getPanAngle()
  set camtlt = getCameraTilt()

  -- sendStatus("position: " & camvec && "   direction: " &
camdir)
  -- sendStatus("x:" & camvec.x && "y:" & camvec.z && "z:" &
camvec.y && "Pan:" & campan && "Tilt:" & camtlt)
  sendStatus("X:" && camvec.x && "Y:" && -camvec.z && "Z:"
&& camvec.y && "pan:" && campan && "tilt:" && camtlt)

end

on jumpToURL url_resource, target
  -- opens a webpage to the specified URL
  gotoNetPage(url_resource, target)

end
```

AV Systems

```
-- AV System
--
-- av_object_list: the complete list of scene dependant AV
objects for the presentation
-- av_object:  Reference to the tied AV sprite

global av_object_list
global av_object
global av_atts
global topsprite
global currentscene

on initializeAV av_idx
  -- initializes the AV object for a timeline synched
presentation by list number
  -- av_object_list = [[#name:"postal",
#attributes:[#filepath: "track2.rm", #mediatype:
#realplayer, #size:[#x:0,#y:0,#w:0,#h:0]]]]
  sendStatus("initializing audio/video object")


  -- get the attributes
  av_atts = av_object_list[av_idx].attributes


  -- create the av sprite
  av_object = topsprite
  topsprite = topsprite + 1

  sprite(av_object).puppet = TRUE
  case av_atts.mediatype of
    #realplayer:
      sendStatus("building real player instance")
      sprite(av_object).member = member("RealPlayer") --
associate the RM member
      -- associate the target behavior to the sprite
      targetScript = script("RealMedia Target").new()
      --
sprite(av_object).scriptInstanceList.add(targetScript)
      -- call( #beginSprite, [targetScript] )


  end case
  sprite(av_object).rect = rect(av_atts.size.x,
av_atts.size.y, (av_atts.size.x + av_atts.size.w),
(av_atts.size.y + av_atts.size.h))


  -- check the filepath
  sendStatus("verifying av filepath")
  -- load the object
  sprite(av_object).member.filename = av_atts.filepath
  sendStatus("loading av file" &&
sprite(av_object).member.filename)

  -- play/pause for object information
  -- ("reading av file information")

  -- pass the information to the system.
  -- sendStatus("file status: " &
sprite(av_object).getMediaStatus())
  updateStage
  avPlay()
end

on avPlayPause
  -- if the video is playing, pauses it.  if the video is
paused, plays it.

  -- check to make sure there is a registered av object
  if not(voidP(av_object)) then -- object is initialized
    -- check the mediatype
    case (av_atts.mediatype) of
      #realplayer:
        -- double check for #realmedia type
        if sprite(av_object).member.type = #realmedia then
          if sprite(av_object).mediaStatus = #playing then
            sprite(av_object).pause()
          else if sprite(av_object).mediaStatus = #paused
then
            sprite(av_object).play()
          end if
          sendStatus("PlayPause activated")
        end if

    end case
  end if

end

on avGetDuration
  -- returns the duration of the av object in milliseconds
  if not(voidP(av_object)) then
    case(av_atts.mediatype) of
      #realplayer:
        if sprite(av_object).member.type = #realmedia then
          return sprite(av_object).duration
        end if
    end case
  end if
  return VOID
end

on avPlay
```

```
  -- plays the video
  if not(voidP(av_object)) then
    case(av_atts.mediatype) of
      #realplayer:
        if sprite(av_object).member.type = #realmedia then
          sprite(av_object).play()
        end if

    end case
  end if
end

on avCurrentTime
  -- returns the current time of the av object in
milliseconds
  -- returns the duration of the av object in milliseconds

  case av_atts.mediatype of
    #realplayer:
      if sprite(av_object).member.type = #realmedia then
        return sprite(av_object).currentTime
      end if

  end case


end

on av_setExternalTimeline av_name, av_region
  -- sets the external timeline for a current scene to the
object
  currentscene.settings.timeline.type = #external
  currentscene.settings.timeline.attributes = [#name:
av_name]
  set av_map = getRegion(av_region)
  if voidP(av_map) then
    puppetSprite(topsprite, TRUE)
    av_object = topsprite
    topsprite = topsprite + 1
    sprite(av_object).rect = rect(0,0,0,0)
  else
    puppetSprite(av_map.spritenum, TRUE)
    av_object = av_map.spritenum
    sprite(av_object).rect = rect(av_map.x, av_map.y,
(av_map.x + av_map.w), (av_map.y + av_map.h))

  end if

end
```

3D SCENE SETUP SCRIPTS

```
on enterFrame me
  -- these commands are held until the 3D scene is
completely loaded.
  -- 3D Sprite first displays on this page, ensuring all
textures have
  -- correctly loaded.

  sendStatus("Initializing 3D system")
  -- reset the event queue
  resetEventQueue()

  -- reset the navigational settings
  resetNavSettings()

  -- set the overall scene properties
  setWorldParameters()

  -- reset the camera & camera object
  createCameraObject()

  -- create textures for camera fades
  createCameraTextures()

  -- run the preworld_events
  XML_ParsePreworld()

  -- set the initial camera object position
  setupCameraLocation()

  -- run the first or current scene
  XML_LoadCurrentScene()

  -- prepare and start the command loop
  initializeTimeline()
end
```

XML Loading

```
global XML_doc
global XML_presets
global XML_member
global XML_state

on loadXML xml_url
  sendStatus("Loading XML file")

  -- takes either a path file or a full URL, and loads the
XML directions.

  XML_doc = new(xtra "xmlparser")
  XML_state = FALSE

  set XMLpath = VOID
  set XMLtype = VOID

  set modetest = the runmode



  case (modetest) of:
    "Author":
      XMLpath = the moviePath & "test_file.xml"
      XMLtype = #local
    "Projector":
      XMLpath = the moviePath & "test_file.xml"
      XMLtype = #local
    "Plugin":
      XMLpath = ""
      XMLtype = #remote
      if externalParamCount() > 0 then
        XMLPath = externalParamValue("sw2")
        sendstatus("XML Path: " & xmlpath)
      end if
  end case

  if (XMLtype = #local) then
    -- create the temporary member to hold the text
    XMLfile = new(#text)

    -- load in the file, and use in the database
    tempIn = new(xtra "FileIO")
    tempIn.openFile(XMLpath, 1)
    XMLfile.text = CRLF2CR(tempIn.readFile())
    tempIn.closeFile()
    tempIn = 0

    XML_member = XMLfile -- set the member reference

    errCode = XML_doc.parseString(XMLfile.text)
    errorString = XML_doc.getError()
    if voidP(errorString) then
      XML_state = TRUE
    else
      alert "XML Parse Error: " & errorString
      -- Exit from the handler
      exit
    end if

    XML_TopLevelParse()

  else if (XMLtype = #remote) then
    errCode = XML_doc.parseURL(XMLpath,
#XML_RemoteLoadComplete)
  end if




end

on XML_RemoteLoadComplete
  if voidP(XML_doc.getError()) then
    XML_state = TRUE
  else
    alert "XML Parse Error: " & XML_doc.getError()
    exit
  end if

  XML_TopLevelParse()
end


on XML_TopLevelParse
  -- parse the top level xml

  if (XML_state) then
    -- loop through the top level
    repeat with xi = 1 to XML_doc.child[1].child.count
      case (XML_doc.child[1].child[xi].name) of:
        "setup":
          XML_ParseSetup(XML_doc.child[1].child[xi])
        "body":
          XML_ParseBody(XML_doc.child[1].child[xi])
      end case
    end repeat
  end if

  -- start the presentation
  -- go "3dBegin"

end
```

XML Load Verification

```
on exitFrame me
  -- test the XML readiness
  global XML_state

  if NOT(XML_state) then
    go to the frame
  end if

end
```

XML Parsing (Setup Phase)

```
global XML_doc
global currentscene
global user_keymap
global user_speedsettings
global scenemodels
global av_object_list

on XML_ParseSetup xml_resource
  -- sends the files to the appropriate spot
  -- loop through the top level
  repeat with ri = 1 to xml_resource.child.count
    case (xml_resource.child[ri].name) of:
      "preferences":
        XML_ParsePreferences(xml_resource.child[ri])
      "worldlist":
        XML_ParseWorldlist(xml_resource.child[ri])
      "layout":
        XML_ParseLayout(xml_resource.child[ri])
      "avlist":
        XML_ParseAVList(xml_resource.child[ri])
    end case
  end repeat
end


on XML_ParsePreferences xml_resource
  -- adjusts the settings to those set out by the XML
document
  repeat with si = 1 to xml_resource.child.count
    case (xml_resource.child[si].name) of:
      "navigation":
        XML_ParseNavigation(xml_resource.child[si])
      "keymap":
        XML_ParseKeymap(xml_resource.child[si])
      "speed":
        XML_ParseSpeed(xml_resource.child[si])
    end case
  end repeat
end


on XML_ParseLayout xml_resource
  -- creates the region layouts by adding regions

  repeat with si = 1 to xml_resource.child.count
    case (xml_resource.child[si].name) of:
      "region": -- only valid tag so far
        -- set default values
        set rname = ""
        set xval = 0
        set yval = 0
        set wval = 0
        set hval = 0

        repeat with ai = 1 to
xml_resource.child[si].attributeName.count
          att_name =
xml_resource.child[si].attributeName[ai]
          att_val =
xml_resource.child[si].attributeValue[ai]
          case (att_name) of:
            "top":
              if integerP(att_val.integer) then xval =
att_val.integer
            "left":
              if integerP(att_val.integer) then yval =
att_val.integer
            "width":
              if integerP(att_val.integer) then wval =
att_val.integer
            "height":
              if integerP(att_val.integer) then hval =
att_val.integer
            "name":
              rname = att_val

          end case

        end repeat

        -- final verify
        if (stringP(rname) = 1) AND (rname <> "") then
          addRegion(rname, xval, yval, wval, hval)
        end if

    end case
  end repeat
end

on XML_ParseWorldlist xml_resource
  -- adds the listed model references to the scenemodels
address
  repeat with si = 1 to xml_resource.child.count

    set datalist = [#name:"", #filepath:""]

    case xml_resource.child[si].name of:
      "world":

        repeat with ai = 1 to
xml_resource.child[si].attributeName.count
          att_name =
xml_resource.child[si].attributeName[ai]
          att_val =
xml_resource.child[si].attributeValue[ai]
```

```
          case (att_name) of:
            "name":
              if stringP(att_val) then datalist.name =
att_val
            "path":
              if stringP(att_val) then
                -- test if this is an http:// call or just a
relative path
                --                if (att_val starts
"http://") then
                --                  datalist.path = att_val
                --                else
                --                  datalist.path = the
moviePath & att_val
                --                end if
                datalist.filepath = the moviePath & att_val
              end if
          end case

        end repeat

        -- check the list before adding it
        if (datalist.name <> "") AND (datalist.filepath <>
"") then
          scenemodels.add(datalist)
        end if

    end case
  end repeat
end

on XML_ParseAVList xml_resource
  -- parses the list of av objects and adds to the system

  repeat with si = 1 to xml_resource.child.count

    set datalist = [#name:"", #path:"", #type:#undefined]

    case xml_resource.child[si].name of:
      "av-object":
        repeat with ai = 1 to
xml_resource.child[si].attributeName.count
          att_name =
xml_resource.child[si].attributeName[ai]
          att_val =
xml_resource.child[si].attributeValue[ai]

          case (att_name) of:
            "name":
              if stringP(att_val) then datalist.name =
att_val
            "path":
              if stringP(att_val) then
```

```
                -- test if this is an http:// call or just a
relative path
                --               if (att_val starts
"http://") then
                --                 datalist.path = att_val
                --               else
                --                 datalist.path = the
moviePath & att_val
                --               end if
                datalist.path = the moviePath & att_val
              end if
            "type":
              case (att_val) of:
                "realplayer": datalist.type = #realplayer
                "realmedia": datalist.type = #realplayer
              end case

          end case

        end repeat

        -- check the list before adding it
        if (datalist.name <> "") AND (datalist.path <> "")
AND (datalist <> #undefined) then
          set av_list_item = [#name: datalist.name,
#attributes:[#filepath: datalist.path, #mediatype:
datalist.type, #size:[#x:0, #y:0, #h:0, #w:0]]]
          put "adding item " & av_list_item
          av_object_list.add(av_list_item)
        end if

    end case
  end repeat
end

on XML_ParseNavigation xml_resource
  -- adjusts currentsettings.settings.navigation

  -- build a temp object
  set navsettings = currentscene.settings.navigation

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "enabled":
        case att_val of:
          "true": navsettings.enabled = TRUE
          "false": navsettings.enabled = FALSE
        end case

      "type":
        case att_val of:
          "free": navsettings.type = #free
        end case

      "mouse_enabled":
        case att_val of:
          "true": navsettings.mouse.enabled = TRUE
          "false": navsettings.mouse.enabled = FALSE
        end case

      "mouse_buffer":
        if floatP(att_val.float) then
          navsettings.mouse.buffer_percent = att_val.float
        end if

      "restrict_zaxis":
        case att_val of:
          "true": navsettings.restrictions.zaxis = TRUE
          "false": navsettings.restrictions.zaxis = FALSE
        end case

      "boundary":
        case att_val of:
          "true": navsettings.boundary.enabled = TRUE
          "false": navsettings.boundary.enabled = FALSE
        end case

      "proximity":
        case att_val of:
          "true": navsettings.position.enabled = TRUE
          "false": navsettings.position.enabled = FALSE
        end case

      "collision":
        case att_val of:
          "true": navsettings.collision.enabled = TRUE
          "false": navsettings.collision.enabled = FALSE
        end case

    end case
  end repeat

  -- update the current settings
  currentscene.settings.navigation = navsettings


end


on XML_ParseKeymap xml_resource
  -- adjusts user_keymap

  -- build a temp object
  set keymap = user_keymap

  repeat with ai = 1 to xml_resource.attributeName.count
```

```
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "moveforward":
        if integerP(att_val.integer) then
keymap.move_forward = att_val.integer
      "movebackward":
        if integerP(att_val.integer) then
keymap.move_backward = att_val.integer
      "moveleft":
        if integerP(att_val.integer) then keymap.move_left =
att_val.integer
      "moveright":
        if integerP(att_val.integer) then keymap.move_right
= att_val.integer
      "turnup":
        if integerP(att_val.integer) then keymap.rotate_up =
att_val.integer
      "turndown":
        if integerP(att_val.integer) then keymap.rotate_down
= att_val.integer
      "turnleft":
        if integerP(att_val.integer) then keymap.rotate_left
= att_val.integer
      "turnright":
        if integerP(att_val.integer) then
keymap.rotate_right = att_val.integer
      "mouselook":
        if integerP(att_val.integer) then keymap.mouse_look
= att_val.integer
    end case

  end repeat

  -- update the keymap
  user_keymap = keymap

end

on XML_ParseSpeed xml_resource
  -- adjusts user_speedsettings

  -- build a temp object
  set speed = user_speedsettings

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "keymove":
        if floatP(att_val.float) then speed.move =
att_val.float
      "keyrotate":
        if floatP(att_val.float) then speed.rotate =
att_val.float
      "mouserotate":
        if floatP(att_val.float) then speed.mouse_speed =
att_val.float
      "invertmouse":
        case att_val of:
          "true": speed.mouse_invert = TRUE
          "false": speed.mouse_invert = FALSE
        end case

    end case
  end repeat

  -- update the speed settings
  user_speedsettings = speed

end
```

XML Parsing Scripts (Presentation Phase)

```
global XML_doc
global currentscene
global user_keymap
global user_speedsettings
global scenemodels
global sceneLayoutProps
global xml_worldsettings
global xml_preworld
global xml_scenelist
global diagnostic
global current_scene_id
global base_scene_id

on XML_ParseBody xml_resource
  -- get the overall settings & parse the scenes

  -- get the world settings
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of :
      "world":
        if stringP(att_val) AND (att_val <> "") then
          -- get the scene name
          currentscene.scenename = att_val
        end if
      "region":
        if stringP(att_val) AND (att_val <> "") then
          -- set the region definition for the 3D object
          set region_props = getRegion(att_val)
          if voidP(region_props) then
            sendStatus("Unrecognized region name: " &
att_val)
          else
            -- update the scene props
            sceneLayoutProps.x = region_props.x
            sceneLayoutProps.y = region_props.y
            sceneLayoutProps.w = region_props.w
            sceneLayoutProps.h = region_props.h
          end if

        end if
      "diagnostic":
        case (att_val) of:
          "true": diagnostic = TRUE
          "false": diagnostic = FALSE
        end case
      "base":
        if stringP(att_val) and (att_val <> "") then
          -- set this to the base scene global
          base_scene_id = att_val
        end if

    end case


  end repeat

  -- initialize the XML scene object
  xml_scenelist = [:]

  repeat with ri = 1 to xml_resource.child.count
    case (xml_resource.child[ri].name) of:
      "worldsettings":
        xml_worldsettings = xml_resource.child[ri]
        XML_ParseWorldSettings(xml_worldsettings)
      "preworld":
        xml_preworld = xml_resource.child[ri]
      "scene":
        -- get the name of the scene
        scenename = ""
        repeat with ai = 1 to
xml_resource.child[ri].attributeName.count
          case (xml_resource.child[ri].attributeName[ai])
of:
            "name":
              scenename =
xml_resource.child[ri].attributeValue[ai]
          end case
        end repeat

        -- replace or update the scene
        if stringP(scenename) AND (scenename <> "") then
          if voidP(getaProp(xml_scenelist, scenename)) then
            xml_scenelist.addProp(scenename,
xml_resource.child[ri])
          else
            -- already exists, update XML entry
            xml_scenelist[scenename] =
xml_resource.child[ri]
          end if
        end if

    end case
  end repeat
end

on XML_ParseWorldSettings xml_resource
  -- updates the currentscene for this particular world.
Takes the <worldsettings> tags & enclosed objects
  -- at this point, the currentscene has already been set,
so it is safe to
  -- write in the new model, lighting, and camera properties
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "inherit":
        case (att_val) of:
          "true":
currentscene.settings.model.worldsettings.structure.preserve
_structure = TRUE
          "false":
currentscene.settings.model.worldsettings.structure.preserve
_structure = FALSE
        end case
      "background":
        if (checkHexColor(att_val) <> 0) then

currentscene.settings.lighting.worldsettings.background_colo
r = att_val
        end if
      "ambient":
        if (checkHexColor(att_val) <> 0) then

currentscene.settings.lighting.worldsettings.ambient_color =
att_val
        end if
    end case
  end repeat

  -- sort the worldsettings
  repeat with ci = 1 to xml_resource.child.count
    case xml_resource.child[ci].name of:
      "position":
        XML_ParseWorldPosition(xml_resource.child[ci])
      "rotation":
        XML_ParseWorldRotation(xml_resource.child[ci])
      "scale":
        XML_ParseWorldScale(xml_resource.child[ci])
      "highlight":
        XML_ParseHighlightSettings(xml_resource.child[ci])
      "camera":
        XML_ParseCameraSettings(xml_resource.child[ci])
      "headlight":
        XML_ParseHeadlightSettings(xml_resource.child[ci])
    end case
  end repeat

end

on XML_ParseWorldPosition xml_resource
  -- set the currentscene world position settings
  new_position =
currentscene.settings.model.worldsettings.orientation.positi
on
```

```
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "x":
        if floatP(att_val.float) then new_position.x =
att_val.float
      "y":
        if floatP(att_val.float) then new_position.z = -
att_val.float
      "z":
        if floatP(att_val.float) then new_position.y =
att_val.float
    end case
  end repeat
  -- update the position object

currentscene.settings.model.worldsettings.orientation.positi
on = new_position
end

on XML_ParseWorldRotation xml_resource
  -- sets the current scene world rotation settings
  new_rotation =
currentscene.settings.model.worldsettings.orientation.rotati
on

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "x":
        if floatP(att_val.float) then new_rotation.x =
att_val.float
      "y":
        if floatP(att_val.float) then new_rotation.y =
att_val.float
      "z":
        if floatP(att_val.float) then new_rotation.z =
att_val.float
    end case
  end repeat

  -- update the rotation object

currentscene.settings.model.worldsettings.orientation.rotati
on = new_rotation
end

on XML_ParseWorldScale xml_resource
  -- set the current world scale settings
  new_scale =
currentscene.settings.model.worldsettings.orientation.scale

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "x":
        if floatP(att_val.float) then new_scale.x = 0.01 *
(att_val.float)  -- percentage
      "y":
        if floatP(att_val.float) then new_scale.y = 0.01 *
(att_val.float)
      "z":
        if floatP(att_val.float) then new_scale.z = 0.01 *
(att_val.float)
    end case
  end repeat

  -- update the scale object

currentscene.settings.model.worldsettings.orientation.scale
= new_scale

end

on XML_ParseHighlightSettings xml_resource
  -- updates the current world highlighting settings
  new_highlight =
currentscene.settings.model.worldsettings.highlight

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "time":
        if integerP(att_val.integer) then
new_highlight.cycle_duration = att_val.integer
      "color":
        if (checkHexColor(att_val) <> 0) then
new_highlight.diffuse = rgb(att_val)
      "ambient":
        if (checkHexColor(att_val) <> 0) then
new_highlight.ambient = rgb(att_val)
    end case
  end repeat

  -- update the current scene settings
  currentscene.settings.model.worldsettings.highlight =
new_highlight

end
```

```
on XML_ParseCameraSettings xml_resource
  -- updates the camera settings with the preworld settings
  new_camera = currentscene.settings.camera.initial_position

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "x":
        if floatP(att_val.float) then new_camera.x_pos =
att_val.float
      "y":
        if floatP(att_val.float) then new_camera.z_pos = -
att_val.float
      "z":
        if floatP(att_val.float) then new_camera.y_pos =
att_val.float
      "pan":
        if floatP(att_val.float) then new_camera.pan_deg =
att_val.float
      "tilt":
        set tilt_angle = att_val.float
        if floatP(tilt_angle) then
          if (tilt_angle >= -90) AND (tilt_angle <= 90) then
-- test for value
            new_camera.tilt_deg = tilt_angle
          end if
        end if
      "fadecolor":
        if (checkHexColor(att_val) <> 0) then
currentscene.settings.camera.fading.default_color = att_val
    end case
  end repeat
  -- update the object
  currentscene.settings.camera.initial_position = new_camera
end

on XML_ParseHeadlightSettings xml_resource
  -- updates the camera headlight with the new data
  new_headlight = currentscene.settings.camera.headlight

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "enabled":
        case (att_val) of:
          "true": new_headlight.enabled = TRUE
          "false": new_headlight.enabled = FALSE
        end case
      "type":
        case (att_val) of:
          "spot": new_headlight.type = #spot
          "directional": new_headlight.type = #directional
        end case
      "color":
        if (checkHexColor(att_val) <> 0) then
new_headlight.color = rgb(att_val)

    end case
  end repeat

end

on XML_ParsePreworld
  -- tells the system to parse a series of prescene commands
  XML_ParsePrescene(xml_preworld)
end

on XML_ParsePrescene xml_resource
  -- reads through the passed XML (from preworld or
prescene) and
  -- executes the commands in the order passed.

  set commandtype = #prescene

  repeat with ci = 1 to xml_resource.child.count
    case (xml_resource.child[ci].name) of:
      "camerafade":
        XML_ParseCameraFade(xml_resource.child[ci],
#prescene)
      "disinclude":
        XML_ParseDisinclude(xml_resource.child[ci])
      "group":
        XML_ParseGroup(xml_resource.child[ci])
      "ungroup":
        XML_ParseUngroup(xml_resource.child[ci])
      "toggle":
        XML_ParseToggle(xml_resource.child[ci], #prescene)
      "lightcolor":
        XML_ParseLightColor(xml_resource.child[ci],
#prescene)
      "setcolor":
        XML_ParseSetColor(xml_resource.child[ci], #prescene)
      "resetcolor":
        XML_ParseResetColor(xml_resource.child[ci],
#prescene)
      "setopacity":
        XML_ParseSetOpacity(xml_resource.child[ci],
#prescene)
      "set-position":
        XML_ParseSetObjectPosition(xml_resource.child[ci],
#object, #prescene)
      "set-camera-position":
        XML_ParseSetObjectPosition(xml_resource.child[ci],
#camera, #prescene)
      "rotate-object":
        XML_ParseRotateObject(xml_resource.child[ci], FALSE,
#prescene)
      "set-object-orientation":
        XML_ParseRotateObject(xml_resource.child[ci], TRUE,
#prescene)
      "reset-object-orientation":
        XML_ParseResetOrientation(xml_resource.child[ci])
      "set-click-trigger":
        XML_ParseSetClickTrigger(xml_resource.child[ci],
#prescene)
      "reset-click-trigger":
        XML_ParseResetClickTrigger(xml_resource.child[ci],
#prescene)
      "highlight-object":
        XML_ParseHighlightObject(xml_resource.child[ci],
#prescene)
      "unhighlight-object":
        XML_ParseHighlightObject(xml_resource.child[ci],
#prescene)
      "trigger":
        XML_ParseTrigger(xml_resource.child[ci], #prescene)
      "set-proximity-trigger":
        XML_ParseSetProximity(xml_resource.child[ci],
#prescene)
      "reset-proximity-trigger":
        XML_ParseResetProximity(xml_resource.child[ci],
#prescene)
      "web-link":
        XML_ParseWeblink(xml_resource.child[ci])
    end case
  end repeat
end

on XML_ParseScene xml_resource
  sendStatus("parsing scene" &&
xml_resource.attributeValue["name"])
  -- parses the items contained within this current scene
  set av_object = VOID
  set av_region = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "avobject":  if stringP(att_val) then av_object =
att_val
      "avregion": if stringP(att_val) then av_region =
att_val
    end case
  end repeat
  if NOT(voidP(av_object)) then
    -- there is an external timeline object.  Use this.
    av_setExternalTimeline(av_object, av_region)


  end if


  -- MISSING CODE HERE - CHANGE THE TIMELINE TO THIS AV
OBJECT

  repeat with ci = 1 to xml_resource.child.count
    if (xml_resource.child[ci].name = "prescene") then
      XML_ParsePrescene(xml_resource.child[ci])
    else
      XML_ParseObject(xml_resource.child[ci])
    end if
  end repeat
end

on XML_LoadCurrentScene
  -- loads the scene associated with the current_scene_id
  -- a void current_scene_id defaults to loading the
base_scene_id.
  -- In case of a VOID base_scene_id, re-runs the preworld
and sets the timeline to internal

  set xml_scenename = ""

  if voidP(xml_scenelist.getaProp(current_scene_id)) then
    if voidP(xml_scenelist.getaProp(base_scene_id)) then
      -- if neither is defined, use the first item in the
xml scenelist
      if (xml_scenelist.count > 0) then
        base_scene_id = 1
        current_scene_id = base_scene_id
        XML_ParseScene(xml_scenelist[current_scene_id])
      else
        -- nothing found - just float there
        sendStatus("current & base scenes not defined")
        currentscene.settings.timeline.type = #internal
        currentscene.settings.timeline.start = 0
      end if

    else
      current_scene_id = base_scene_id
      XML_ParseScene(xml_scenelist[current_scene_id])
    end if
  else
    XML_ParseScene(xml_scenelist[current_scene_id])
  end if

end
```

```
on XML_ParseObject xml_resource
  -- determines the type of object and runs the appropriate
parser
  case (xml_resource.name) of:
    "camerafade": XML_ParseCameraFade(xml_resource, #live)
    "toggle": XML_ParseToggle(xml_resource, #live)
    "lightcolor": XML_ParseLightColor(xml_resource, #live)
    "setcolor": XML_ParseSetColor(xml_resource, #live)
    "resetcolor": XML_ParseResetColor(xml_resource, #live)
    "setopacity": XML_ParseSetOpacity(xml_resource, #live)
    "point-camera-at-location":
XML_ParseCameraPoint(xml_resource, #location, #live)
    "point-camera-at-object":
XML_ParseCameraPoint(xml_resource, #object, #live)
    "point-camera-at-direction":
XML_ParseCameraPoint(xml_resource, #direction, #live)
    "point-camera-at-path":
XML_ParseCameraPoint(xml_resource, #path, #live)
    "point-camera-at-none":
XML_ParseCameraPoint(xml_resource, #none, #live)
    "point-object-at-location":
XML_ParseObjectPoint(xml_resource, #location, #live)
    "point-object-at-object":
XML_ParseObjectPoint(xml_resource, #object, #live)
    "point-object-at-direction":
XML_ParseObjectPoint(xml_resource, #direction, #live)
    "point-object-at-path":
XML_ParseObjectPoint(xml_resource, #path, #live)
    "point-object-at-none":
XML_ParseObjectPoint(xml_resource, #none, #live)
    "object-motion": XML_ParseMotionSequence(xml_resource,
#live)
    "camera-motion": XML_ParseMotionSequence(xml_resource,
#live)
    "set-position": XML_ParseSetObjectPosition(xml_resource,
#object, #live)
    "set-camera-position":
XML_ParseSetObjectPosition(xml_resource, #camera, #live)
    "rotate-object": XML_ParseRotateObject(xml_resource,
FALSE, #live)
    "set-object-orientation":
XML_ParseRotateObject(xml_resource, TRUE, #live)
    "reset-object-orientation":
XML_ParseResetOrientation(xml_resource)
    "rotate-camera": XML_ParseRotateCamera(xml_resource,
#live)
    "adjust-camera": XML_ParseCameraPanTilt(xml_resource,
#live)
    "set-click-trigger":
XML_ParseSetClickTrigger(xml_resource, #live)
      "reset-click-trigger":
XML_ParseResetClickTrigger(xml_resource, #live)
    "highlight-object":
XML_ParseHighlightObject(xml_resource, #live)
```

```
    "unhighlight-object":
XML_ParseHighlightObject(xml_resource, #live)
    "trigger": XML_ParseTrigger(xml_resource, #live)
    "set-proximity-trigger":
XML_ParseSetProximity(xml_resource, #live)
    "reset-proximity-trigger":
XML_ParseResetProximity(xml_resource, #live)
      "web-link": XML_ParseWeblink(xml_resource)
  end case
end
```

XML Parse Command Tags (Setup)

```
-- these scripts parse the setup-style (non-queue) commands

on XML_ParseDisinclude xml_resource
  set obj_type = ""
  set obj_name = ""

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "object":
        if stringP(att_val) then obj_name = att_val
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
          "light": obj_type = #light

        end case

    end case

  end repeat

  if (obj_type = "") then
    obj_type = verifyObject(att_val)
  end if

  case (obj_type) of:
    #model:
      DisincludeModel(obj_name)
    #group:
      DisincludeGroup(obj_name)
    #light:
      DisincludeLight(obj_name)
  end case

end

on XML_ParseGroup xml_resource
  -- builds a group out of a list of child objects
  set groupname = ""

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "name":
        if stringP(att_val) then groupname = att_val
    end case
  end repeat

  if (groupname <> "") then
    -- this is a valid group name - process the sub-objects
    createNewGroup(groupname)

    repeat with ci = 1 to xml_resource.child.count
      case (xml_resource.child[ci].name) of:
        "object":
          set obj_name = ""
          set obj_type = ""

          repeat with ai = 1 to
xml_resource.child[ci].attributeName.count
            att_name =
xml_resource.child[ci].attributeName[ai]
            att_val =
xml_resource.child[ci].attributeValue[ai]

            case (att_name) of:
              "name":
                if stringP(att_val) then obj_name = att_val
              "type":
                case (att_val) of:
                  "model": obj_type = #model
                  "group": obj_type = #group
                  "light": obj_type = #light
                end case

            end case

          end repeat

          -- if no type has been determined, go get one
          if (obj_type = "") then obj_type =
verifyObject(obj_name)

          -- send the command
          case (obj_type) of:
            #model:
              addModelToGroup(groupname, obj_name)
            #light:
              addLightToGroup(groupname, obj_name)
          end case



      end case
    end repeat
  end if


end
```

```
on XML_ParseUngroup xml_resource
  -- if a group exists, this will break it up into it's
individual objects
  set groupname = ""

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "name":
        if stringP(att_val) then groupname = att_val
    end case

  end repeat

  if (groupname <> "") then
    destroyGroup(groupname)
  end if

end

on XML_ParseHighlightObject xml_resource
  -- tells an object to highlight or unhighlight
  set highlight_direction = xml_resource.name
  set obj_name = VOID
  set obj_duration = #indefinite

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "duration":
        if integerP(att_val.integer) then
          obj_duration = att_val.integer
        end if

    end case
  end repeat

  if NOT(voidP(obj_name)) AND NOT(voidP(obj_duration)) then
    set queue_object = XML_GetCommandAtts(xml_resource)

    case (highlight_direction) of:
      "highlight-object":
        queue_object.command = #highlight_object
        queue_object.duration = 1
        queue_object.attributes = [#object_name: obj_name]
        addEventToQueue(queue_object)
        cancel_object = duplicate(queue_object)
        -- check for a cancel event
        if integerP(obj_duration) then
          cancel_object.name = queue_object.name & "_off"
          cancel_object.starttime = queue_object.starttime +
obj_duration
          cancel_object.command = #reset_highlight
          addEventToQueue(cancel_object)
        end if

      "unhighlight-object":
        queue_object.command = #reset_highlight
        queue_object.duration = 1
        queue_object.attributes = [#object_name: obj_name]
        addEventToQueue(queue_object)
    end case
  end if
end
```

XML Parse Command Tags (Queue)

```
-- these commands will add events to the queue if called.
-- Events denoted as stage #prescene will be executed
without a timeline call
-- #live events will be added to the queue.

on XML_GetCommandAtts xml_resource
  -- returns a set of the common atts to build an event from
command items
  -- RETURNS:
  -- [#name: unique event name (for targets & holds)
  --  #command: #none
  --  #starttime:  time(ms) OR #now OR #hold
  --  #duration: time(ms)
  --  #priority: #verify, #noverify
  --  #attributes: [:]
  --  #hold_trigger: #none OR targetname

  -- sets the default name to the current milliseconds
  -- (this keeps alternate commands from being confused)

  -- build the default atts
  set command_atts = [#name: the milliseconds & "_" &
random(1028) , #command: #none, #starttime:0, #duration:0,
#priority: #noverify, #attributes: [:], #hold_trigger:
#none]

  -- retrieve any attributes located in the XML tag
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "name":
        if stringP(att_val) then command_atts.name = att_val
      "start":
        if integerP(att_val.integer) then
          command_atts.starttime = att_val.integer
        else if (att_val = "now") then
          command_atts.starttime = #now
        else if (att_val = "hold") then
          command_atts.starttime = #hold
        end if
      "duration":
        if integerP(att_val.integer) then
command_atts.duration = att_val.integer
      "verify":
        case (att_val) of:
          "true":  command_atts.priority = #verify
          "false": command_atts.priority = #noverify
        end case
      "holdfor":
        if stringP(att_val) then command_atts.hold_trigger =
att_val
    end case
  end repeat

  return command_atts

end




on XML_ParseCameraFade xml_resource, event_stage
  set fade_direction = VOID

  -- get the fade attributes
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "direction":
        case (att_val) of:
          "in": fade_direction = #in
          "out": fade_direction = #out
        end case
    end case

  end repeat

  -- check for valid required atts
  if NOT(voidP(fade_direction)) then

    set fade_atts = [#overlay: #default, #direction:
fade_direction]

    if (event_stage = #prescene) then
      -- prescene - execute the command
      CameraFade(fade_atts, 1)
    else
      -- this is a live exercise
      -- get the standard attributes
      camera_atts = XML_GetCommandAtts(xml_resource)
      camera_atts.command = #fade_camera

      camera_atts.attributes = fade_atts

      addEventToQueue(camera_atts)
    end if

  end if

end

on XML_ParseToggle xml_resource, event_stage
```

```
  -- toggles a model or light on/off.   (MISSING CODE HERE -
GROUPS DO NOT CURRENTLY WORK)
  set toggle_direction = VOID
  set object_name = VOID
  set object_type = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
object_name = att_val
      "type":
        case (att_val) of:
          "model": object_type = #model
          "group": object_type = #group
          "light": object_type = #light
        end case

      "direction":
        case (att_val) of:
          "on": toggle_direction = #on
          "off": toggle_direction = #off
        end case
    end case
  end repeat

  if NOT(voidP(toggle_direction)) AND
NOT(voidP(object_name)) then
    -- verify the object type
    if voidP(object_type) then object_type =
verifyObject(object_name)

    -- is this prescene?
    case (event_stage) of:
      #prescene:
        case (object_type) of:
          #model:
            case (toggle_direction) of:
              #on: TurnOnModel(object_name)
              #off: TurnOffModel(object_name)
            end case
          #light:
            case (toggle_direction) of:
              #on: TurnOnLight(object_name)
              #off: TurnOffLight(object_name)
            end case
          #group:
            case (toggle_direction) of:
              #on: TurnOnGroup(object_name)
              #off: TurnOffGroup(object_name)
            end case
        end case

      #live:
        -- get the command atts
        toggle_atts = XML_GetCommandAtts(xml_resource)

        case (object_type) of:
          #model:
            toggle_atts.command = #toggle_model
            toggle_atts.attributes = [#model_name:
object_name, #toggle: toggle_direction]
            addEventToQueue(toggle_atts)
          #light:
            toggle_atts.command = #toggle_light
            toggle_atts.attributes = [#light_name:
object_name, #toggle: toggle_direction]
            addEventToQueue(toggle_atts)
          #group:
            toggle_atts.command = #toggle_group
            toggle_atts.attributes = [#group_name:
object_name, #toggle: toggle_direction]
            addEventToQueue(toggle_atts)
        end case
    end case
  end if
end

on XML_ParseLightColor xml_resource, event_stage
  -- changes a light color  (for a light="background" value,
changes the background color)

  set light_name = VOID
  set light_color = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "color":
        light_color = checkHexColor(att_val)
        if (light_color = 0) then light_color = VOID
      "light":
        if stringP(att_val) AND (att_val <> "") then
          if (att_val = "background") then
            light_name = #background
          else
            light_name = att_val
          end if
        end if
    end case
  end repeat

  if NOT(voidP(light_color)) AND NOT(voidP(light_name)) then
    case (event_stage) of:
      #prescene:
        changeLightColor([#light_name: light_name,
#start_value: #this, #end_value: rgb(light_color)], 1)
      #live:

        light_atts = XML_GetCommandAtts(xml_resource)
        light_atts.command = #change_lightcolor
        light_atts.attributes = [#light_name: light_name,
#start_value: #this, #end_value: rgb(light_color)]
        addEventToQueue(light_atts)

    end case

  end if

end

on XML_ParseSetColor xml_resource, event_stage
  -- sets the color of a model, group, or light object

  set obj_name = VOID
  set obj_type = VOID
  set obj_color = VOID
  set amb_color = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "color":
        obj_color = checkHexColor(att_val)
        if (obj_color = 0) then obj_color = VOID
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
          "light": obj_type = #light
        end case
      "ambient":
        amb_color = checkHexColor(att_val)
        if (amb_color = 0) then amb_color = VOID
    end case
  end repeat

  -- if this is a valid object
  if NOT(voidP(obj_name)) AND NOT(voidP(obj_color)) then
    -- set the type, if not already
    if (voidP(obj_type)) then
      obj_type = verifyObject(obj_name)
      -- if still void, check for the background object
      if (voidP(obj_type)) then
```

```
        if (obj_name = "background") then
          obj_name = #background
          obj_type = #light
        end if
      end if
    end if

    -- set a default ambient color
    if amb_color = VOID then amb_color = "#000000"

    case (obj_type) of:
      #light:
        case (event_stage) of:
          #prescene:
            changeLightColor([#light_name: obj_name,
#start_value: #this, #end_value: rgb(obj_color)], 1)
          #live:
            light_atts = XML_GetCommandAtts(xml_resource)
            light_atts.command = #change_lightcolor
            light_atts.attributes = [#light_name: obj_name,
#start_value: #this, #end_value: rgb(obj_color)]
            addEventToQueue(light_atts)
        end case

      #model:
        case (event_stage) of:
          #prescene:
            changeModelColor([#model_name: obj_name,
#start_value: #this, #end_value:[#ambient: rgb(amb_color),
#diffuse: rgb(obj_color)]], 1)
          #live:
            model_atts = XML_GetCommandAtts(xml_resource)
            model_atts.command = #change_modelcolor
            model_atts.attributes = [#model_name: obj_name,
#start_value: #this, #end_value: [#ambient: rgb(amb_color),
#diffuse: rgb(obj_color)]]
            addEventToQueue(model_atts)
        end case

      #group:
        case (event_stage) of:
          #prescene:
            changeGroupColor([#group_name: obj_name,
#start_value: #this, #end_value:[#ambient: rgb(amb_color),
#diffuse: rgb(obj_color)]], 1)
          #live:
            group_atts = XML_GetCommandAtts(xml_resource)
            group_atts.command = #change_groupcolor
            group_atts.attributes = [#group_name: obj_name,
#start_value: #this, #end_value: [#ambient: rgb(amb_color),
#diffuse: rgb(obj_color)]]
            addEventToQueue(group_atts)
        end case
    end case
```

```
  end if

end

on XML_ParseResetColor xml_resource, event_stage
  -- sets the color of a model or group object back to it's
original state

  set obj_name = VOID
  set obj_type = VOID



  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
        end case
    end case
  end repeat

  -- if this is a valid object
  if NOT(voidP(obj_name)) then
    -- set the type, if not already
    if (voidP(obj_type)) then
      obj_type = verifyObject(obj_name)
    end if


    case (obj_type) of:
      #model:
        case (event_stage) of:
          #prescene:
            changeModelColor([#model_name: obj_name,
#start_value: #this, #end_value:#original], 1)
          #live:
            model_atts = XML_GetCommandAtts(xml_resource)
            model_atts.command = #change_modelcolor
            model_atts.attributes = [#model_name: obj_name,
#start_value: #this, #end_value: #original]
            addEventToQueue(model_atts)
        end case

      #group:
        case (event_stage) of:
          #prescene:
            changeGroupColor([#group_name: obj_name,
#start_value: #this, #end_value:#original], 1)
          #live:
            group_atts = XML_GetCommandAtts(xml_resource)
            group_atts.command = #change_groupcolor
            group_atts.attributes = [#group_name: obj_name,
#start_value: #this, #end_value: #original]
            addEventToQueue(group_atts)
        end case
    end case
  end if

end

on XML_ParseSetOpacity xml_resource, event_stage
  -- sets the opacity of a model or group to a percentage
value

  set obj_name = VOID
  set obj_type = VOID
  set obj_percent = VOID


  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
        end case
      "percent":
        if floatP(att_val.float) then obj_percent = 0.01 *
att_val.float
    end case
  end repeat

  -- if this is a valid object
  if NOT(voidP(obj_name)) AND NOT(voidP(obj_percent)) then
    -- set the type, if not already
    if (voidP(obj_type)) then
      obj_type = verifyObject(obj_name)
    end if


    case (obj_type) of:
      #model:
        case (event_stage) of:
          #prescene:
            changeModelOpacity([#model_name: obj_name,
#start_value:#this, #end_value:obj_percent], 1)  --

          #live:
            model_atts = XML_GetCommandAtts(xml_resource)
            model_atts.command = #change_opacity
            model_atts.attributes = [#model_name: obj_name,
#start_value: #this, #end_value: obj_percent]
            addEventToQueue(model_atts)
        end case

      #group:
        case (event_stage) of:
          #prescene:
            changeGroupOpacity([#group_name: obj_name,
#start_value: #this, #end_value:obj_percent], 1)
          #live:
            group_atts = XML_GetCommandAtts(xml_resource)
            group_atts.command = #change_groupopacity
            group_atts.attributes = [#group_name: obj_name,
#start_value: #this, #end_value: obj_percent]
            addEventToQueue(group_atts)
        end case
    end case

  end if
end


on XML_ParseCameraPoint xml_resource, point_type,
event_stage
  -- this is the camera object.  Set the object properties,
then fetch the object.
  set obj_name = "camera"
  set obj_type = #camera

  XML_ParseTargeting(xml_resource, point_type, obj_name,
obj_type, event_stage)
end

on XML_ParseObjectPoint xml_resource, point_type,
event_stage
  -- get the name & type of the object that will be assigned
a target
  set obj_name = VOID
  set obj_type = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
```

```
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
          "light": obj_type = #light
          "camera": obj_type = #camera
        end case
    end case
  end repeat

  if NOT(voidP(obj_name)) then
    XML_ParseTargeting(xml_resource, point_type, obj_name,
obj_type, event_stage)
  end if
end


on XML_ParseTargeting xml_resource, point_type, obj_name,
obj_type, event_stage
  -- takes camera & point objects and parses the targeting
into the event queue
  set target_type = VOID

  case (point_type) of:
    #location:
      target_type = #position
    #direction:
      target_type = #direction
    #object:
      target_type = #object
    #path:
      target_type = #path
    #none:
      target_type = #none
  end case

  if NOT voidP(target_type) then   -- this is a valid target
type
    -- get the command atts (name, start time, duration,
etc)
    command_atts = XML_GetCommandAtts(xml_resource)

    -- get the type attributes
    target_atts = XML_GetTargetAtts(xml_resource,
target_type)

    -- send the target transition command
    targetTransitionSetup(obj_name, command_atts.starttime,
command_atts.duration, target_atts.attributes,
target_atts.accel_time, target_atts.decel_time)
    put "targetTransitionSetup(" && obj_name &&
command_atts.starttime && command_atts.duration &&
target_atts.attributes && target_atts.accel_time &&
target_atts.decel_time &&")"
```

```
    -- is this a duration limited persistant targeting?
    if (target_atts.attributes.persistent = TRUE) AND
(target_atts.persistdur <> #indefinite) then
      if integerP(target_atts.persistdur) then
        targetTransitionSetup(obj_name,
(command_atts.starttime + target_atts.persistdur), 0,
[#type:#none, #data: VOID, #persistent: FALSE], 0, 0)
      end if
    end if
  end if

end




on XML_GetTargetAtts xml_resource, target_type
  -- sets targetTransitionSetups for the event
  set obj_name = VOID
  set xyz_pos = vector(0,0,0)
  set acceltime = #none
  set deceltime = #none
  set trg_persist = FALSE
  set trg_persistdur = 0

  -- fetch the attributes
  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "x": if floatP(att_val.float) then xyz_pos.x =
att_val.float
      "y": if floatP(att_val.float) then xyz_pos.z = -
att_val.float
      "z": if floatP(att_val.float) then xyz_pos.y =
att_val.float
      "target-object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "accel":
        if integerP(att_val.integer) then acceltime =
att_val.integer
      "decel":
        if integerP(att_val.integer) then deceltime =
att_val.integer
      "persistent":
        case (att_val) of:
          "true": trg_persist = TRUE
          "false": trg_persist = FALSE
        end case
      "persist-for":
        if integerP(att_val.integer) AND (att_val.integer >
0) then trg_persistdur = att_val.integer
```

```
        if (att_val = "indefinite") then trg_persistdur =
#indefinite
    end case

  end repeat

  -- return the appropriate data set
  set return_atts = [#attributes:[#type: target_type, #data:
VOID, #persistent: trg_persist], #accel_time:acceltime,
#decel_time: deceltime, #persistdur: trg_persistdur]

  case (target_type) of:
    #position:
      return_atts.attributes.data = xyz_pos
    #direction:
      xyz_pos.normalize()
      return_atts.attributes.data = xyz_pos
    #object:
      return_atts.attributes.data = obj_name
  end case

  return return_atts

end


on XML_ParseSetObjectPosition xml_resource, is_camera,
event_stage
  -- parses a "set-position" or "set-camera-position" XML
command
  -- and if valid positions the specified object, or queues
the positioning.

  if (is_camera = #object) then
    set obj_name = VOID
    set obj_type = VOID

    -- get the object name & type
    repeat with ai = 1 to xml_resource.attributeName.count
      att_name = xml_resource.attributeName[ai]
      att_val = xml_resource.attributeValue[ai]

      case (att_name) of:
        "object": if stringP(att_val) AND (att_val <> "")
then obj_name = att_val
        "type":
          case (att_val) of:
            "model": obj_type = #model
            "group": obj_type = #group
            "light": obj_type = #light

          end case
      end case
    end repeat
    -- if the object type is still not set, verify it
    obj_type = verifyObject(obj_name)
  else
    obj_name = "camera"
    obj_type = #camera
  end if

  -- make sure the object name has been set
  if NOT(voidP(obj_name)) then
    -- get the XYZ attributes
    set new_pos = XML_ReadPoint(xml_resource)



    if vectorP(new_pos) then
      -- this is a valid point.  Position or queue the
object
      case (event_stage) of:
        #prescene:
          case  (obj_type) of:
            #camera: setCameraPosition(new_pos)
            #model: setModelPosition(obj_name, new_pos)
            #group: setGroupPosition(obj_name, new_pos)
            #light: setLightPosition(obj_name, new_pos)
          end case

        #live:
          set obj_atts = [#name: obj_name, #type: obj_type,
#position: new_pos]
          position_atts = XML_GetCommandAtts(xml_resource)
          position_atts.command = #set_position
          position_atts.attributes = obj_atts
          addEventToQueue(position_atts)
      end case

    end if
  end if

end

on XML_ParseWeblink xml_resource
  -- parses a weblink command at runtime
  set url_link = VOID
  set url_target = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    att_name = xml_resource.attributeName[ai]
    att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "href": if stringP(att_val) then url_link = att_val
      "window": if stringP(att_val) then url_target =
att_val
```

```
    end case

  end repeat

  if NOT(voidP(url_link)) then
    jumpToURL(url_link, url_target)
  end if

end
```

LOADFILE

```
global scene          -- we are going to reference the 3D
castmember at least once in this script
global currentscene
global scenemodels


on exitFrame me
  put scene.state

  if check3Dready(scene) then    -- call custom handler:  is
the SW3D castmember ready?

    scene.resetworld()
    --
scene.loadFile(getExternalModelpath(currentscene.scenenum))

scene.loadFile(getExternalModelpath(getNamedListIndex(scenem
odels, currentscene.scenename)))
    go "3dinit"                      -- if it is ready, go to
the "init" marker
  else
    go to the frame                   -- if it is not ready,
wait here
  end if
end
```

XML Trigger Scripts

```
on XML_ParseSetClickTrigger xml_resource, event_stage
  -- parses a <set-click-trigger> command
  set obj_name = VOID
  set obj_cont = TRUE
  set obj_start = VOID
  set obj_dur = #indefinite
  set trigger_name = VOID

  -- get the attributes
  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) and (att_val <> "") then
obj_name = att_val
      "continue":
        case (att_val) of:
          "true": obj_cont = TRUE
          "false": obj_cont = FALSE
        end case
      "start":
        if integerP(att_val.integer) and (att_val >= 0) then
obj_start = att_val.integer
      "duration":
        if integerP(att_val.integer) and (att_val >= 0) then
obj_dur = att_val.integer
      "trigger":
        if stringP(att_val) and (att_val <> "") then
trigger_name = att_val
    end case
  end repeat

  if NOT(voidP(obj_name)) then
    if voidP(obj_start) then
      obj_start = 0 -- start at the beginning
    end if

    -- register the trigger XML
    -- if the trigger is called by name, just use that.
    -- if the trigger is NOT called by name, then
    if voidP(trigger_name) then
      trigger_name = "trigger_" & the milliseconds & " " &
random(100) -- build a unique name
      registerTriggerXML(trigger_name, xml_resource) --
register the XML
    end if


    -- build the event attributes
    trigger_atts = [#name: obj_name, #continue: obj_cont,
#trigger: trigger_name]

    -- build the queue registration event
    set reg_atts = XML_GetCommandAtts(xml_resource)

    reg_atts.starttime = obj_start
    reg_atts.duration = 0
    reg_atts.command = #register_click_event
    reg_atts.attributes = trigger_atts

    addEventToQueue(reg_atts)

    if (obj_dur <> #indefinite) and (integerP(obj_dur)) then
      -- build the cancel event
      set unreg_name = "unreg_" & the milliseconds
      set unreg_time = obj_start + obj_dur

      unreg_atts = [#name: unreg_name, #command:
#unregister_click_event, #starttime: unreg_time,
#duration:1, #priority: #noverify, #attributes:[#name:
obj_name], #hold_trigger:#none]
      addEventToQueue(unreg_atts)
    end if


  end if

end


on XML_ParseTrigger xml_resource, event_stage
  -- reads a named trigger and adds it to the
trigger_xml_list by name.
  set trigger_name = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "name":
        if stringP(att_val) and (att_val <> "") then
trigger_name = att_val
    end case
  end repeat

  if NOT(voidP(trigger_name)) then
    registerTriggerXML(trigger_name, xml_resource)
  end if
end

on XML_ParseSetProximity xml_resource, event_stage
  -- reads a proximity set event, and builds the trigger.
  set obj_name = VOID
  set obj_cont = FALSE
  set obj_start = VOID
  set obj_dur = #indefinite
  set trigger_name = VOID
  set leave_trigger_name = VOID
  set obj_distance = VOID

  -- get the attributes
  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) and (att_val <> "") then
obj_name = att_val
      "continue":
        case (att_val) of:
          "true": obj_cont = TRUE
          "false": obj_cont = FALSE
        end case
      "start":
        if integerP(att_val.integer) and (att_val >= 0) then
obj_start = att_val.integer
      "duration":
        if integerP(att_val.integer) and (att_val >= 0) then
obj_dur = att_val.integer
      "trigger":
        if stringP(att_val) and (att_val <> "") then
trigger_name = att_val
      "leave-trigger":
        if stringP(att_val) and (att_val <> "") then
leave_trigger_name = att_val
      "distance":
        if floatP(att_val.float) then obj_distance =
att_val.float
    end case
  end repeat

  if NOT(voidP(obj_name)) AND NOT(voidP(obj_distance)) then
    if voidP(obj_start) then
      obj_start = 0 -- start at the beginning
    end if

    -- build the event attributes
    trigger_atts = [#name: obj_name, #distance:
obj_distance, #continue: obj_cont, #trigger: [#enter:
trigger_name, #leave: leave_trigger_name]]

    -- build the queue registration event
    set reg_atts = XML_GetCommandAtts(xml_resource)

    reg_atts.starttime = obj_start
    reg_atts.duration = 0
    reg_atts.command = #register_prox_event
    reg_atts.attributes = trigger_atts

    addEventToQueue(reg_atts)

    if (obj_dur <> #indefinite) and (integerP(obj_dur)) then
      -- build the cancel event
      set unreg_name = "unreg_" & the milliseconds
      set unreg_time = obj_start + obj_dur

      unreg_atts = [#name: unreg_name, #command:
#unregister_prox_event, #starttime: unreg_time, #duration:1,
#priority: #noverify, #attributes:[#name: obj_name],
#hold_trigger:#none]
      addEventToQueue(unreg_atts)
    end if


  end if


end

on XML_ParseResetProximity xml_resource, event_stage
  -- registers an #unregister_prox_trigger command for the
specified object
  set obj_name = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) and (att_val <> "") then
obj_name = att_val
    end case
  end repeat

  if NOT(voidP(obj_name)) then
    set unreg_atts = XML_GetCommandAtts(xml_resource)
    unreg_atts.command = #unregister_prox_event
    unreg_atts.duration = 1
    unreg_atts.attributes = [#name: obj_name]
    addEventToQueue(unreg_atts)
  end if


end

on XML_ParseResetClickTrigger xml_resource, event_stage
  -- registers an #unregister_click_event for the specified
object
  set obj_name = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
```

```
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) and (att_val <> "") then
obj_name = att_val
    end case
  end repeat

  if NOT(voidP(obj_name)) then
    set unreg_atts = XML_GetCommandAtts(xml_resource)
    unreg_atts.command = #unregister_click_event
    unreg_atts.duration = 1
    unreg_atts.attributes = [#name: obj_name]
    addEventToQueue(unreg_atts)
  end if

end
```

Trigger Scripts

```
global click_trigger_list
global trigger_xml_list
global scene
global invisible_model_list
global prox_trigger_list
global current_prox_on
global current_prox_off

on registerClickTrigger trigger_attributes
  -- runs the register click trigger event by registering a
click
  -- trigger into the click_trigger_list.  XML to run on
trigger click is
  -- already contained at trigger_xml_list, under a unique
name.
  --
  -- trigger attributes format:
  -- [#name: model name of trigger
  --  #continue: TRUE or FALSE (FALSE expires the trigger
after one click)
  --  #trigger: unique trigger name]

  if voidP(trigger_attributes) then return -1

  set obj_name = trigger_attributes.getaProp(#name)
  set obj_cont = trigger_attributes.getaProp(#continue)
  set obj_trigger = trigger_attributes.getaProp(#trigger)

  if voidP(obj_name) OR voidP(obj_cont) OR
voidP(obj_trigger) then
    return -1
  else
    -- all objects are valid - add this to the list
    set trigger_test = click_trigger_list.getaProp(obj_name)
    if voidP(trigger_test) then
      -- this is a new trigger, insert it
      click_trigger_list.addProp(obj_name, [#trigger:
obj_trigger, #continue: obj_cont])
    else
      -- this is an old trigger, replace it
      click_trigger_list[obj_name] = [#trigger: obj_trigger,
#continue: obj_cont]
    end if

    return TRUE
  end if
end

on unregisterClickTrigger trigger_atts
  -- removes a trigger on a named object
  if listP(trigger_atts) then
```

```
    set model_name = trigger_atts.getaProp(#name)
    resetClickTrigger(model_name)
    return TRUE
  else
    return -1
  end if

end


on resetClickTrigger model_name
  -- removes a trigger from the list
  if voidP(model_name) then
    return -1
  else
    click_trigger_list.deleteProp(model_name)
    return TRUE
  end if
end

on registerTriggerXML trigger_name, trigger_xml
  -- Adds or Replaces the XML associates with the trigger
name in the trigger XML list
  -- trigger_xml_list holds all XML for the trigger calls.

  if voidP(trigger_name) or voidP(trigger_xml) then
    return -1
  end if

  set name_test = trigger_xml_list.getaProp(trigger_name)

  if voidP(name_test) then
    -- this is a new name
    trigger_xml_list.addProp(trigger_name, trigger_xml)
  else
    -- replace the named object
    trigger_xml_list[trigger_name] = trigger_xml
  end if
end

on unregisterTriggerXML trigger_name
  -- removes an XML trigger from the list
  trigger_xml_list.deleteProp(trigger_name)
end

on callClickTrigger model_name
  -- calls a click from a model and calls the associated
trigger

  -- get the trigger info
  set trigger_info = click_trigger_list.getaProp(model_name)
  if listP(trigger_info) then
    -- has returned a list of properties
    set trigger_name = trigger_info.getaProp(#trigger)
    set trigger_continue = trigger_info.getaProp(#continue)

    -- call the trigger
    callTrigger(trigger_name)

    -- erase the event, if one time only
    if (trigger_continue = FALSE) then
      resetClickTrigger(model_name)
    end if
  end if


end




on callTrigger trigger_name
  -- called when a trigger is executed
  -- fetches the associated XML information from the
trigger_xml_list
  -- and parses the child data for runtime.

  if stringP(trigger_name) then
    -- get the XML source
    xml_resource = trigger_xml_list.getaProp(trigger_name)

    if voidP(xml_resource) then
      return -1
    end if

    -- parse the XML
    repeat with ci = 1 to xml_resource.child.count
      XML_ParseObject(xml_resource.child[ci])
    end repeat
  end if
end

on checkClickPoint check_point
  -- checks the models underneath the location of the click
point
  -- for possible trigger clicks.
  --
  -- CheckClickPoint Steps:
  -- ----------------------------
  -- 1. Run detailed modelsUnderLoc on check_point to get
closest model (max 5).  Evaluate.
  -- 2. First one not a shade clone gets the click.

  set model_list =
scene.camera[1].modelsUnderLoc(check_point, 10, #simple)

  -- Find the top visible model
  repeat with mc = 1 to model_list.count
    -- disregard shadeclones
    if NOT(model_list[mc].name contains "_shadeclone") then
      -- disregard invisible models
      if
voidP(invisible_model_list.getaProp(model_list[mc].name))
then
        callClickTrigger(model_list[mc].name)
        -- put "clicktrigger: " & model_list[mc].name
        exit repeat
      end if
    end if
  end repeat

end



on registerProxTrigger trigger_attributes
  -- activates a proximity trigger based on the attributes
passed in.
  -- trigger events are named and stored in the
trigger_xml_list
  --
  -- trigger attributes format:
  -- [#name: model name of trigger
  --  #distance: float value of trigger distance
  --  #continue: TRUE or FALSE (FALSE expires the trigger
after one click)
  --  #trigger: [#enter: trigger1, #leave: trigger2]]

  if voidP(trigger_attributes) then return -1

  set obj_name = trigger_attributes.getaProp(#name)
  set obj_cont = trigger_attributes.getaProp(#continue)
  set obj_trigger = trigger_attributes.getaProp(#trigger)
  set obj_distance = trigger_attributes.getaProp(#distance)

  if voidP(obj_name) OR voidP(obj_cont) OR
voidP(obj_distance) OR NOT(listP(obj_trigger)) then
    return -1
  else
    -- all objects are valid - add this to the list
    set trigger_test = prox_trigger_list.getaProp(obj_name)
    if voidP(trigger_test) then
      -- this is a new trigger, insert it
      prox_trigger_list.addProp(obj_name, [#distance:
obj_distance, #trigger: obj_trigger, #continue: obj_cont])
    else
      -- this is an old trigger, replace it
      prox_trigger_list[obj_name] = [#distnace:
obj_distance, #trigger: obj_trigger, #continue: obj_cont]
    end if
    current_prox_on.deleteOne(obj_name)
    current_prox_off.deleteOne(obj_name)
    current_prox_off.add(obj_name)

    return TRUE
  end if
end

on unregisterProxTrigger trigger_atts
  -- removes a trigger on a named object
  if listP(trigger_atts) then
    set model_name = trigger_atts.getaProp(#name)
    resetProxTrigger(model_name)
    return TRUE
  else
    return -1
  end if

end

on resetProxTrigger model_name
  -- removes a trigger from the list
  if voidP(model_name) then
    return -1
  else
    prox_trigger_list.deleteProp(model_name)
    current_prox_on.deleteOne(model_name)
    current_prox_off.deleteOne(model_name)
    return TRUE
  end if
end

on evaluateProximities
  -- checks the camera position for on & off calls
  --
  -- STEPS:
  -- 1. Check for new ON events (evaluate all in off list)
  -- 2.  If any are now inside the proximity, call the
trigger, set to ON
  -- 3. Check for new OFF events (evaluate all in on list)
  -- 4.  If any are now outside the proximity, call the
leave trigger
  -- 5.  Set to OFF or remove, depending on continue value

  set camera_pos = getCameraPosition()

  if vectorP(camera_pos) then

    -- check for new ON events
    repeat with pr = 1 to current_prox_off.count
      set model_name = current_prox_off[pr]
      set model_pos = getModelPosition(model_name)
      set trigger_atts =
prox_trigger_list.getaProp(model_name)
      if listP(trigger_atts) then
```

```
        -- valid trigger, get ths distance
        prox_dist = trigger_atts.getaProp(#distance)
        if vectorP(model_pos) and floatP(prox_dist) then
          -- valid model, compare the distance
          set this_distance =
camera_pos.distanceTo(model_pos)
          if (this_distance <= prox_dist) then
            -- trigger the on event
            callTrigger(trigger_atts.trigger.enter)
            -- move this to the off category
            current_prox_off.deleteOne(model_name)
            current_prox_on.add(model_name)

            put "Calling On: " & model_name
          end if
        end if
      end if
    end repeat

    -- check for new OFF events
    repeat with pr2 = 1 to current_prox_on.count
      set model_name = current_prox_on[pr2]
      set model_pos = getModelPosition(model_name)
      set trigger_atts =
prox_trigger_list.getaProp(model_name)
      if listP(trigger_atts) then
        -- valid trigger, get ths distance
        prox_dist = trigger_atts.getaProp(#distance)
        if vectorP(model_pos) and floatP(prox_dist) then
          -- valid model, compare the distance
          set this_distance =
camera_pos.distanceTo(model_pos)
          if (this_distance > prox_dist) then
            -- trigger the on event
            callTrigger(trigger_atts.trigger.leave)
            -- move this to the off category
            current_prox_on.deleteOne(model_name)
            put "Calling Off: " & model_name
            if (trigger_atts.continue = TRUE) then
              current_prox_off.add(model_name)
            end if

          end if
        end if
      end if
    end repeat

  end if
end
```

XML Motion Scripts

```
global motion_catalog

on XML_ParseMotionSequence xml_resource, event_stage
  -- takes a motion sequence and parses it into the
motion_catalog to
  -- be held until runtime execution of the MotionSetup()
command

  set obj_name = VOID
  set obj_type = #model
  set accel_time = #none
  set decel_time = #none
  set start_pos = #here
  set start_vec = vector(0,0,0)

  -- determine the object
  case (xml_resource.name) of:
    "camera-motion":
      obj_name = "camera"
      obj_type = #camera
    "object-motion":
      repeat with ai = 1 to xml_resource.attributeName.count
        set att_name = xml_resource.attributeName[ai]
        set att_val = xml_resource.attributeValue[ai]
        case (att_name) of:
          "object":
            if (stringP(att_val)) AND (att_val <> "") then
obj_name = att_val
          "type":
            case (att_val) of:
              "model": obj_type = #model
              "group": obj_type = #group
              "light": obj_type = #light
              "camera": obj_type = #camera
            end case
        end case
      end repeat
  end case

  if NOT(voidP(obj_name)) then
    -- there is a specified object - add this event to the
motion queue

    -- get the command attributes
    command_atts = XML_GetCommandAtts(xml_resource)

    -- get the accel & decel values
    repeat with ai = 1 to xml_resource.attributeName.count
      set att_name = xml_resource.attributeName[ai]
      set att_val = xml_resource.attributeValue[ai]
      case (att_name) of:
        "accel": if integerP(att_val.integer) then
accel_time = att_val.integer
        "decel": if integerP(att_val.integer) then
decel_time = att_val.integer
        "start-position":
          case (att_val) of:
            "here": start_pos = #here
            "position": start_pos = #position
          end case
        "x": if floatP(att_val.float) then start_vec.x =
att_val.float
        "y": if floatP(att_val.float) then start_vec.z = -
att_val.float
        "z": if floatP(att_val.float) then start_vec.y =
att_val.float

      end case
    end repeat

    -- MISSING CODE HERE !!!!!
    -- GET THE MOTION SEQUENCE CONTAINED WITHIN THE
INDIVIDUAL SECTIONS
    -- set motion_sequence = [[#type: #spline_path,
#data:[vector( 310.7916, 525.0666, -1.38699e3 ), vector( -
675.5729, 1.04613e3, -865.0831 ), vector( -49.2461,
1.07685e3, -117.7038 ), vector( 726.9523, 612.1934, -
455.7021 ), vector( -584.1855, 327.9199, -9.91340e2 ) ]]] --
REPLACE THIS!!!
    -- MISSING CODE HERE !!!!!

    -- set the motion sequence
    set motion_sequence = []

    -- get each motion item
    repeat with ci = 1 to xml_resource.child.count
      set motion_item =
XML_ParseMotionItem(xml_resource.child[ci])
      if NOT(voidP(motion_item)) then
motion_sequence.append(motion_item)
    end repeat


    -- replace the starting position with the starting
vector, if not #here
    if (start_pos = #position) then start_pos = start_vec

    -- build the list to hold in the catalog
    event_name = command_atts.name
    event_start = command_atts.starttime
    event_duration = command_atts.duration
    set catalog_entry = [#name: event_name, #object_name:
obj_name, #object_type: obj_type, #start_time:event_start,
#duration: event_duration, #start_pos: start_pos, #accel:
```

```
accel_time, #decel: decel_time,
#motion_sequence:motion_sequence]

    -- add to the catalog
    setaProp motion_catalog, event_name, catalog_entry

    -- add the placeholder to the queue
    command_atts.command = #call_motion
    command_atts.duration = 0
    command_atts.attributes = [#name: event_name]
    addEventToQueue(command_atts)

  end if
end




on callMotionFromCatalog event_attributes
  -- calls a named motion sequence from the catalog and adds
it to the event queue during runtime
  set event_name = event_attributes.name

  set motion_event = motion_catalog.getaProp(event_name)

  if voidP(motion_event) then
    return -1
  else
    -- add this event to the motion through the
MotionSetup() command

    MotionSetup(motion_event.object_name,
motion_event.object_type, motion_event.start_time,
motion_event.duration, motion_event.start_pos,
motion_event.motion_sequence, motion_event.accel,
motion_event.decel)
    return TRUE
  end if
end

on XML_ParseMotionItem xml_resource
  -- parses commands within the object motion & camera
motion tags.
  -- returns the motion sequence item
  set motion_item = VOID

  case (xml_resource.name) of:
    "direct":
      motion_item = XML_ParseDirectMotion(xml_resource)
    "relative":
      motion_item = XML_ParseRelativeMotion(xml_resource)
    "orbit":
      motion_item = XML_ParseOrbitMotion(xml_resource)
    "path":
      motion_item = XML_ParsePathMotion(xml_resource)
  end case

  return motion_item
end



on XML_ParseDirectMotion xml_resource
  -- reads a "<direct/>" motion tag and returns the motion
sequence item.
  -- item format: [#type: #linear_absolute, #data: <xyz-
vector>]
  -- a valid X, Y, & Z Coordinate is necessary
  set point_vector = XML_ReadPoint(xml_resource)
  if voidP(point_vector) then
    return VOID
  else
    return [#type: #linear_absolute, #data: point_vector]
  end if

end

on XML_ParseRelativeMotion xml_resource
  -- reads a "<relative/>" motion tag and returns the motion
sequence item.
  -- item format: [#type: #linear_relative, #data: <xyz-
vector>]
  -- a valid X, Y, & Z coordinate is necessary
  set point_vector = XML_ReadPoint(xml_resource)
  if voidP(point_vector) then
    return VOID
  else
    return [#type: #linear_relative, #data: point_vector]
  end if
end

on XML_ParseOrbitMotion xml_resource
  -- reads a <orbit></orbit> tag and returns the motion
sequence item
  -- item format: [#type: #orbit_absolute, #data:
[#angle:<float>, #axis:<xyz vector>, #pivot:<xyz vector>]]
  -- a valid X, Y, & Z coordinate is necessary
  set o_pivot = VOID
  set o_angle = VOID
  set o_axis = VOID
  set obj_name = VOID
  set final_pivot = VOID
  set final_axis = VOID
  set rotate_object = FALSE

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "pivot":
        case (att_val) of:
          "object": o_pivot = #object
          "point": o_pivot = #point
        end case
      "angle":
        if floatP(att_val.float) then o_angle =
att_val.float
      "axis":
        case (att_val) of
          "custom": o_axis = #custom
          "x-axis": o_axis = vector(1,0,0)
          "y-axis": o_axis = vector(0,0,-1)
          "z-axis": o_axis = vector(0,1,0)
        end case
      "pivot-object":
        if (stringP(att_val)) and (att_val <> "") then
obj_name = att_val
      "rotate":
        case (att_val) of:
          "true": rotate_object = TRUE
          "false": rotate_object = FALSE
        end case

    end case
  end repeat

  -- if those are void, don't even bother
  if voidP(o_pivot) OR voidP(o_angle) OR voidP(o_axis) then
    return VOID
  else
    set custom_pivot = VOID
    set custom_axis = VOID

    -- get custom attributes, if needed
    repeat with ci = 1 to xml_resource.child.count
      case (xml_resource.child[ci].name) of:
        "pivot": custom_pivot =
XML_ReadPoint(xml_resource.child[ci])
        "axis": custom_axis =
XML_ReadPoint(xml_resource.child[ci])
      end case
    end repeat

    -- set the pivot point
    case (o_pivot) of
      #object: final_pivot = GetObjectPosition(obj_name)
      #point: final_pivot = custom_pivot
    end case

    if (o_axis = #custom) then
      final_axis = custom_axis
    else
```

```
      final_axis = o_axis
    end if

    if vectorP(final_axis) AND vectorP(final_pivot) then

      -- check for a negative rotation angle.
      if (o_angle < 0) then
        o_angle = -o_angle -- rotate the angle
        final_axis = -final_axis -- flip the axis
      end if


      set motion_item = [#type: #orbit_absolute, #data:
[#angle: o_angle, #pivot: final_pivot, #axis: final_axis,
#rotate: rotate_object]]
      return motion_item
    else
      return VOID
    end if

  end if
end

on XML_ParsePathMotion xml_resource
  -- reads the point children of the <path></path> tags and
returns a motion sequence item
  -- item format: [#type: #spline_path, #data: [<xyz-
vector>, ...]
  -- for each point, a valid XYZ coordinate is required, or
it will not be added
  -- to the path checkpoint list.
  set spline_points = []
  repeat with ci = 1 to xml_resource.child.count
    case (xml_resource.child[ci].name) of:
      "point":
        set checkpoint =
XML_ReadPoint(xml_resource.child[ci])
        if NOT(voidP(checkpoint)) then
spline_points.append(checkpoint)
    end case
  end repeat

  return [#type: #spline_path, #data: spline_points]

end


on XML_ReadPoint xml_resource
  -- takes an XML tag and looks for "x", "y", & "z"
attributes
  -- returns an XYZ vector, or VOID if one or more
attributes is missing or invalid
  -- Corrects for Director axis flip
  set xval = VOID
  set yval = VOID
  set zval = VOID
  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "x": if floatP(att_val.float) then xval =
att_val.float
      "y": if floatP(att_val.float) then yval =
att_val.float
      "z": if floatP(att_val.float) then zval =
att_val.float
    end case
  end repeat

  if voidP(xval) OR voidP(yval) OR voidP(zval) then
    return VOID
  else
    -- build the vector
    return vector(xval, zval, -yval)
  end if

end

on XML_ParseRotateObject xml_resource, orientation,
event_stage
  -- reads an XML rotate object event and parses it for the
proper options.
  -- "orientation": TRUE or FALSE -- TRUE indicates that
this object will only have
  --                          it's point-at-orientation moved
about the axis.
  set obj_name = VOID
  set obj_type = VOID
  set obj_ref = VOID
  set obj_angle = VOID
  set obj_axis = VOID
  set final_axis = VOID
  set obj_accel = 0
  set obj_decel = 0

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]

    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
      "type":
        case (att_val) of:
          "model": obj_type = #model
          "group": obj_type = #group
          "light": obj_type = #light
        end case
      "reference":
        case (att_val) of :
          "world": obj_ref = #world
          "self": obj_ref = #self
        end case
      "angle":
        if floatP(att_val.float) then obj_angle =
att_val.float
      "axis":
        case (att_val) of:
          "custom": obj_axis = #custom
          "x-axis": obj_axis = vector(1,0,0)
          "y-axis": obj_axis = vector(0,0,-1)
          "z-axis": obj_axis = vector(0,1,0)
        end case
      "accel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_accel = att_val.integer
      "decel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_decel = att_val
    end case
  end repeat



  -- is this an orientation object?
  if (orientation = TRUE) then obj_ref = #orientation

  -- make sure the values have been set
  if voidP(obj_name) or voidP(obj_ref) or voidP(obj_angle)
or voidP(obj_axis) then
    put "error parsing rotate-object command"
    return -1
  end if

  if (obj_axis = #custom) then
    -- get the custom axis
    set custom_axis = VOID
    repeat with ci = 1 to xml_resource.child.count
      case (xml_resource.child[ci].name) of:
        "axis": custom_axis =
XML_ReadPoint(xml_resource.child[ci])
      end case
    end repeat
    final_axis = custom_axis
  else
    final_axis = obj_axis
  end if

  if vectorP(final_axis) then
    -- this is a valid axis, set the rotation event
    command_atts = XML_GetCommandAtts(xml_resource)
    if (event_stage = #prescene) then
      command_atts.starttime = 0
      command_atts.duration = 0
      obj_accel = 0
      obj_decel = 0
    end if

    rotation_atts = [#type: obj_ref, #axis: final_axis,
#angle: obj_angle]
    rotationSetup(command_atts.name, obj_name, obj_type,
command_atts.starttime, command_atts.duration,
rotation_atts, obj_accel, obj_decel,
command_atts.hold_trigger, false)
    -- rotationSetup object_name, start_time, duration,
rotation_atts, accel_time, decel_time, event_hold,
return_event

  end if
end

on XML_ParseResetOrientation xml_resource
  -- queues the event to reset the orientation for the
object
  set obj_name = VOID

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "object":
        if stringP(att_val) AND (att_val <> "") then
obj_name = att_val
    end case
  end repeat

  set command_atts = XML_GetCommandAtts(xml_resource)
  command_atts.command = #reset_orientation
  command_atts.attributes = [#object_name: obj_name]
  addEventToQueue(command_atts)
end

on XML_ParseRotateCamera xml_resource, event_stage
  -- sets up a camera rotation animation around a set axis
  set obj_name = "camera"
  set obj_type = #camera
  set obj_angle = VOID
  set obj_axis = VOID
  set final_axis = VOID
  set obj_accel = 0
  set obj_decel = 0

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "angle":
        if floatP(att_val.float) then obj_angle =
att_val.float
      "axis":
        case (att_val) of:
          "custom": obj_axis = #custom
          "x-axis": obj_axis = vector(1,0,0)
          "y-axis": obj_axis = vector(0,0,-1)
          "z-axis": obj_axis = vector(0,1,0)
        end case
      "accel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_accel = att_val.integer
      "decel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_decel = att_val
    end case
  end repeat

  -- make sure the values have been set
  if voidP(obj_name) or voidP(obj_angle) or voidP(obj_axis)
then
    put "error parsing rotate-camera command"
    return -1
  end if

  if (obj_axis = #custom) then
    -- get the custom axis
    set custom_axis = VOID
    repeat with ci = 1 to xml_resource.child.count
      case (xml_resource.child[ci].name) of:
        "axis": custom_axis =
XML_ReadPoint(xml_resource.child[ci])
      end case
    end repeat
    final_axis = custom_axis
  else
    final_axis = obj_axis
  end if

  if vectorP(final_axis) then
    -- this is a valid axis, set the rotation event
    command_atts = XML_GetCommandAtts(xml_resource)
    if (event_stage = #prescene) then
      command_atts.starttime = 0
      command_atts.duration = 0
      obj_accel = 0
      obj_decel = 0
    end if

    rotation_atts = [#type: #camera_direction, #axis:
final_axis, #angle: obj_angle]
```

```
    rotationSetup(command_atts.name, obj_name, obj_type,
command_atts.starttime, command_atts.duration,
rotation_atts, obj_accel, obj_decel,
command_atts.hold_trigger, false)
    -- rotationSetup object_name, start_time, duration,
rotation_atts, accel_time, decel_time, event_hold,
return_event

  end if
end

on  XML_ParseCameraPanTilt xml_resource, event_stage
  -- performs a relative pan / tilt animation on the camera
  set camera_pan = 0
  set camera_tilt = 0
  set obj_accel = 0
  set obj_decel = 0

  repeat with ai = 1 to xml_resource.attributeName.count
    set att_name = xml_resource.attributeName[ai]
    set att_val = xml_resource.attributeValue[ai]
    case (att_name) of:
      "pan": if floatP(att_val.float) then camera_pan = -1 *
att_val.float
      "tilt": if floatP(att_val.float) then camera_tilt =
att_val.float
      "accel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_accel = att_val.integer
      "decel":
        if integerP(att_val.integer) AND (att_val.integer >=
0) then obj_decel = att_val.integer
    end case
  end repeat

  command_atts = XML_GetCommandAtts(xml_resource)
  rotation_atts = [#type: #camera_angle, #pan: camera_pan,
#tilt: camera_tilt]
  rotationSetup(command_atts.name, "camera", #camera,
command_atts.starttime, command_atts.duration,
rotation_atts, obj_accel, obj_decel,
command_atts.hold_trigger, false)
end
```

Region Map Scripts

```
global region_map
global topsprite

on addRegion rname, xval, yval, wval, hval
  -- adds a region to the region_map list.
  -- overwrites an already existing name (for on the fly
region changes)

  set newsprite = topsprite
  set datalist = [:]
  if voidP(region_map.getaProp(rname)) then
    -- add the new data object
    datalist = [#x: xval, #y: yval, #w: wval, #h: hval,
#spritenum: newsprite]
    region_map.addProp(rname, datalist)
    topsprite = topsprite + 1
  else
    -- already exists, update the information
    region_map[rname].x = xval
    region_map[rname].y = yval
    region_map[rname].w = wval
    region_map[rname].h = hval
  end if

end

on getRegion rname
  -- returns VOID if rname does not exist, else returns the
properties of this object
  return region_map.getaProp(rname)
end

on setRegion rname
  -- puppets the sprite associated with the region, and
returns the sprite num
  set this_sprite = getRegion(rname)
  if NOT(voidP(this_sprite)) then
    set this_spritenum = this_sprite.spritenum
    puppetSprite(this_spritenum)
    return this_spritenum
  end if
end
```