Authoring and Delivering Mixed Reality Experiences

Matthew O'Connor
Media Convergence Laboratory
Institute for Simulation and Training
University of Central Florida

Charles E. Hughes
Media Convergence Laboratory
School of Computer Science
University of Central Florida

Keywords: Mixed reality, scripting, interactive experiences.

ABSTRACT

Mixed Reality (MR) offers a unique challenge in integrating interacting agents, show-control devices, graphics and audio presentation, and human interaction into a single consolidated system. While each component may be addressed individually, combining their various functionalities via a dynamic script that delivers an interactive, non-linear story (scenario, world) requires a robust process and system. In this paper, we present a scripting engine and an XML-based language that allows an author to interface human interaction with the other elements of an MR experience. The engine and language allow for a wide range of functionality, including real-time binding of variables, interaction with external engines (such as graphics and audio) and sensors/actuators (such as tracking and special effects), a pluggable interface for autonomous interacting agents, a simple integrated physics engine, and a common architecture for expressing actions that have graphical and auditory representations. We exhibit the functionality of flexibility of our system through two major scenarios: a training exercise and an exploratory experience that we demonstrated at a science center in fall, 2004.

1. INTRODUCTION

Current systems for Mixed Reality authoring are suitable for creating visually-based augmented reality applications. In particular, the AMIRE system [1] provides a family of components that address issues such as tracking, registration, video capture and rendering. Their audio library is focused on delivery of standard clips, not on 3d immersion. This system has served as the basis for creating user-friendly environments for authoring specific application from within the real world that is being augmented [3][8].

Panda3D provides a scene graph-based rendering system along with a procedural authoring environment. This procedural language includes commands for manipulating the scene graph, sending and receiving messages, playing audio files, scheduling tasks and controlling behaviors by finite state machines and Python methods [7].

In contrast to the capabilities provided by these and other existing systems, our goal is a system that supports the creation and delivery of the full range of Mixed and Virtual Reality experiences from augmented reality to augmented virtuality to pure virtual; from mixed audio-visual-SFX to pure audio to pure visual; from reactive and preplanned behaviors to AI-based behaviors. Our initial approach, prototyped in various versions during 2002 and 2003 [4], provided the design of a system comprised of cooperating engines (story, graphics, audio and special effects), with the story engine being the central component, the only one that deals with the semantics of actions – the other engines primarily provide services, such as visualization and auralization (Figure 1). These earlier story engines required that experiences be authored in Java, whereas the system reported here uses an agent-based approach, with behaviors describable in XML scripts or via AI plug-ins. Finally, our older systems, while providing platform independent story and special effects engines, were tied to Linux for graphics and to Windows for audio. All engines in the current version run on Linux and Windows. We also believe that MacOS compatibility will be easy to achieve, although we have not tested this as yet.

The focus of this paper is on the story/scripting engine. Separate papers deal with the audio engine and the process, rather than the technology, of creating stories.

2. THE MR SCRIPTING ENGINE

2.1 Overview

The MR Scripting Engine (MRScript hereafter) is a platform for developing experiences that involve multiple interactive system elements. These elements include, but are not limited to, graphics, audio, and show-control devices. MR Script's objective is to provide a linking, synchronizing/control, and agent/actor support mechanism for utilizing these elements in an interactive environment. Environments that are candidates for using MRScipt include those delivering interactive story-telling experiences, technology demonstrations, games, and educational or training programs. The experiences can involve full or partial mixed-reality, but do not necessarily have to be visual – pure aural experiences can and have been developed/delivered with MRScript. The basic structure of MRScript allows multiple elements to connect and be utilized for the realization of any number of such environments, whether shared or stand-alone.

Besides linking and synchronizing diverse sensory components, MRScript operates as the nerve center for all artificial intelligence, scripted, and user-interactive behavior. It is the focal point for a user's interactions, and allows environments and experiences to be developed using any level of artificial intelligence or scripting that is appropriate to achieve a particular goal. The simplest means by which MRScript provides behaviors is through its scripting language. This language was originally conceived to support only the simplest of pre-scheduled and reactive behaviors, but it has already been used to successfully perform surprisingly complex tasks, ones that we had originally expected would be handled by AI plug-ins. This flexibility was possible due to the language’s easy extensibility, which was a primary design goal.

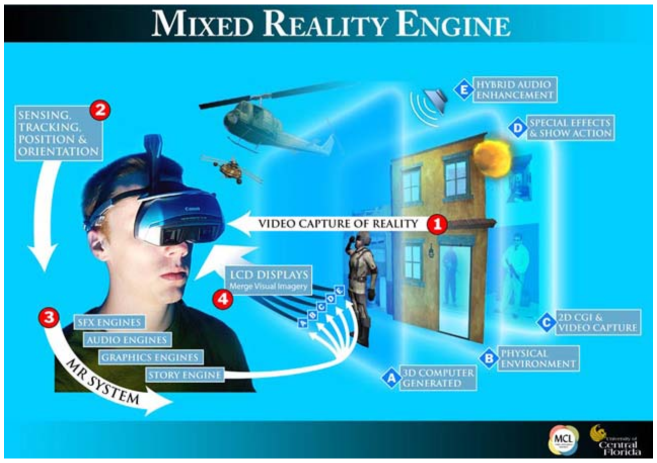

Figure 1. Integration of MR system

2.2 Engine Specifics

An experience that is delivered by MRScript starts when all the system elements required to support the given experience connect to the scripting engine. Once these elements connect, the experience loaded by the engine may begin to take form. This happens with direct interaction between MRScript and its client elements; each client element can be communicated with explicitly, or in broadcast communication, and each can reply appropriately to MRScript with necessary information or user interactions.

2.2.1 Components

MRScript is composed of various parts: the main engine, a physics engine, a real-time dynamic reference resolution system, an Advanced Scripting Language interface (ASL), and client networking support. The elemental construct that the engine is based on is interacting agents. Agents are the definitive elements for a given script. Each agent contains intrinsic elements, such as physics characteristics, that allow for a common interface with the system. Agents may or may not have representation on external client elements that connect during runtime (for example, an agent may exist as a sound on one client element, but have no graphical representation – this is perfectly valid).

Within agents, behaviors of three flavors are available: actions, triggers, and reflexes. Each agent also has a set of state variables, which describes the agent's state at any given time (these are akin to static variables in a normal programming context, the only real difference being that state variables are normally updated once at the end of each iteration, rather than at the moment of each assignment or mathematical operation). Agents may also execute initialization commands for new client elements that connect. These are explicitly defined in an initialization block.

The script is used to define agents for the scripting engine. Scripts may be composed of several component script files, which may include other component files and be included by the main script file. This allows components to be generated and dropped in for usage in any given script. Scripts generally define not only agents, but regional information that can be used with the physical location of agents to provide semantic value to their location, or enable/disable features of certain agents, etc.

2.2.2 Engine Operation

The engine operates on an iterative basis. That is, operations on state variables, transmission of signals, etc, is carried out once all agents have operated. Therefore, there are two stages to the operation of the engine during normal running conditions: *update*, and *operate*. The *update* stage forces state variables to update their values to whatever changed values were stored in the midst of the last operation cycle; it also copies signals that were queued up during the most recent operating cycle to an operating queue that is used during the next iteration (note that this ensures that signals received during one iteration must wait for the next iteration before they can be acted upon). Finally, the update stage makes internal system elements such as the system clock current. The *operate* stage causes agents to evaluate any signals received, to evaluate any triggers that have fired, and to operate reflexes (triggers and reflexes are discussed in the next section). At the end of this stage, the agent's physics attributes are subject to the physics engine, which considers such things as linear and angular velocity/acceleration when updating the agent's physical characteristics. See Table 1 for attributes that can be controlled by the physics engine.

Various special signals are used to notify agents of certain system activities. The START signal is transmitted to all agents when the system starts iterating. The STOP signal is transmitted when a stop is requested, after which agents have five iterations to act on the signal before iterating ceases. Various other signals are used to signal particular agents (the SYSTEM agent is one such agent) of events, such as mouse-movement, button presses, etc.

3. RUNNING AN EXPERIENCE

3.1 Interacting with Client Elements

The system can be operated in one of two modes: GUI and Console. The GUI mode displays an interactive graphical user interface that can be used to start, stop, pause, step, and communicate with the system (or any connected client elements). The Console mode starts system iteration as soon as all specified client elements have connected. Note that no a priori number of elements is required in order for iteration to occur in either the GUI or Console modes.

Console mode has no direct interactive system, so the script is basically left to its own devices when the system is run in this manner. The motivation for this was deployment of experiences, where the system would be required to operate completely autonomously.

Table 1. Settable attributes controlled by physics engine.

Element	Parts	Units
`.location`	`.x .y .z`	mm
`.orientation`	`.yaw .pitch .roll`	°
`.linearVelocity`	`.x .y .z`	mm/s
`.linearAcceleration`	`.x .y .z`	mm/s^2
`.angularVelocity`	`.yaw .pitch .roll`	°/s
`.angularAcceleration`	`.yaw .pitch .roll`	°/s^2

3.2 Scripting Fundamentals

Communication among agents is one of the fundamental requirements for a system involving interacting agents. The goal is, of course, to allow a collection of agents to act as a whole, or interact in an interesting manner. To fully understand how this is accomplished in MRScript, we will explore how agents are created by the script writer.

Agents are defined by state variables, actions, triggers and reflexes. The three behaviors (*actions*, *triggers*, *reflexes*) all may contain multiple cases, each of which include *guards*, *results*, and *commands*. A guard ensures that certain state variable conditions are met before the case is allowed to evaluate its results or commands. A case may have zero guards, in which case its results and commands are always evaluated. Results are used to change state variable values, either using explicit setting of values, or arithmetic operations and expressions. Commands are the communication mechanism for the agent, in the scripting language: communiques are the only form of communication among the agents themselves, whereas graphics, audio and show-control commands are used to trigger actions on connected client elements (for example, an agent's action may cause a movie to play on a graphics element). A third class, whose members are known as system commands, are available for scripting-engine-specific interaction (such as debug enabling). Advanced Scripting Language (ASL) blocks are considered to be results, due to their functionality. Details of ASL are given later in this paper.

Actions are behaviors that must be explicitly triggered by a signal. Signals (known as *communiques* in the scripting language) are the primary method for agent interaction, both among themselves and with the engine itself (the engine has an intrinsic *system* agent that allows us to use a common interface for invoking its activities). Once an action has been triggered by a signal, it is evaluated the next iteration that the agent is operated upon.

Triggers are based on the operating time of the experience, as maintained by the scripting engine. A trigger is tripped when its time value is exceeded by the scripting engine's system clock (this clock tells time relative to the start of system iteration – if the system becomes paused, the clock also pauses and continues precisely where it left off when the system starts running again). Triggers can be reset automatically, or be one-shot triggers that must wait until the system is restarted before they can be used again. The time value they are based on can be dynamic.

Reflexes operate during every iteration, and can be used to check for certain state variable conditions; this is the motivation for their existence. Once a reflex's guards are all enabled (evaluate to true), its results and commands are executed. This is perhaps the most expensive behavior element of the three, since extensive evaluation may take place. However, it can be extremely useful for scanning for changes in state, or updating state variables per some formula, etc.

3.3 The Advanced Scripting Language (ASL)

ASL was inspired by the need for extended functionality, such as flow-control, real-time updating and evaluating of state variables (i.e., in the midst of a single iteration), and loops. ASL uses a very rudimentary form of the language structure found in C and Java. It allows the use of all references, dynamic state variable reference generation (described below) and all interaction capable with the normal scripting constructs.

The programming elements ASL defines are: if, for, goto and assignment (including assignments such as +=, *=, etc). While it is not meant to be a replacement for the scripting language itself, it embodies almost all the functionality of the main scripting language, and provides additional flexibility and extensibility to the language. All ASL blocks must be housed in cases, and are therefore dependent on the operation of an action, trigger or reflex. So while the ASL is a powerful, flexible element of the scripting language, it cannot operate on its own.

3.4 Engine Support Features

The MRScript has several features that enhance not only the script-writers abilities, but also enhance operation of the scripts themselves.

3.4.1 System Calls

Before ASL was defined, complex functions such as distance calculation, etc, were done using system calls. These still exist, and still have usage; their implementation is pure Java, and so they operate much faster than the doubly-interpreted ASL code (it is an interpreted language inside an interpreted language).

System calls may take arguments in the form of variables or text, and generally return some form of state variable containing the result sought. For example, the distance calculation %getDistance will return a state variable that contains the distance between two locations arguments provided in its call.

3.4.2 Dynamic References

One of the most powerful aspects of the scripting language is the ability to use real-time dynamic reference resolution. A reference is a scripting language construct that refers to a particular state variable, returning the value contained within the variable itself. For example, a state variable foo may contain the text "bar". The syntax {@foo} in the appropriate expression/text will cause {@foo} to be replaced with "bar" when the expression/text is evaluated. Instances where this activity is most easily observed occur in communiques: an agent may refer to itself using {@SELF} to obtain its own name, which can then be used in conjunction with other command keywords to perform some function on a client engine (example: "{@SELF} show model" for an agent named Foo will cause "Foo show model" to be transmitted to some client element – perhaps a graphics client is the intended recipient).

The true power of this can be witnessed in the ability to form state variable names from other references. For example, if an agent contains the state variable foo = "bar", and another state variable bar1 = "10", the result of {@{@foo}1} will be {@bar1}, which will then be resolved to bar1's value: "10". All references are resolved at run-time, and are therefore completely dynamic. If foo's value were to change in the above example, the change would be reflected in the next evaluation of {@{@foo}1}. This resolution capability is available in both normal scripting and ASL.

3.4.3 Resolution System and Caching

All references must eventually be resolved into basic state variables. {@{@foo}1} contains not only an internal reference to foo, but (with foo = "bar") is a reference to bar1. The internal reference is termed monotonic, since it can be directly linked to a state variable. The whole reference is a compound reference, since it is composed of an internal reference – the fact that it also contains additional text is irrelevant. The construct {@{@foo}} --> {@bar} is perfectly valid.

Since all references must eventually break down into monotonic references, these references can be cached. Also, textual elements that contain a single monotonic state variable reference (a very common occurrence) are also cached, since they are effectively returning only the contents of a state variable instead of the state variable itself. During some tests of a complex script (one containing over 100 agents and heavily utilizing not only ASL but the full capability of dynamic resolution), caching of known monotonic state variables and text containing only a single monotonic state variable showed between 60-70% cache hits, which resulted in a significant decrease in the time spent resolving the state variables themselves. The cache system uses two hash maps: one for monotonic state variables, and one for textual-based monotonic references.

3.5 ASL Virtual Machine

The ASL is compiled at runtime and executed when required using the ASL Virtual Machine. This is a very simple machine that has a program counter and two status flags. All references are resolved using the real-time dynamic resolution system (the caching system provides an enormous benefit to ASL in this regard). There are only a few operations available within the ASLVM; these are compare, assign, jump and no-op. Instructions are executed synchronously, and only one ASL block is executed at a time. At present, this limited VM's capabilities seem sufficient to provide the functionality of the C/Java-style language constructs described above.

The virtual machine throws a Machine Exception if dynamic resolution fails or an operand's syntax is invalid. This allows graceful failure of the system to occur via the try … catch mechanism of Java.

3.6 AI Plug-in

Agents can use various artificial intelligence (A.I.) algorithms available through a common, plug-in interface. This interface encapsulates the algorithm, insulating its implementation details from the client agent. An agent interacts with a given algorithm by instantiating it as a type of component, passing it parameters either by explicit call or implicit value modification, and activating it either by system call (explicit) or on an automatic per-iteration basis (implicit).

The following algorithms may be considered for in-clusion:

A set of fuzzy logic math functions may be used on any number of agent state variables. The functions may operate ignorant of the semantic value of their results. Agents can set up fuzzy logic curves, either by parameter or by truth - table specification.

An IHO (feedback optional) neural net maps I input nodes through H hidden nodes and results in O outputs. The inputs and outputs can be represented as real numbers, integer numbers, or Boolean values. Neural net training may take place with back propagation, which should occur off-line (outside normal simulation operation). However, it may be useful to provide a back propagation training function for real-time operation, whereby expected outputs are provided and adjusted within the network. All interfaces with the network should take place at its inputs and outputs only. Internal net structure should not be visible to the agent.

An ANT algorithm may be used for navigation, but might also be (experimentally) applied to behaviors. Given a choice of paths, an ANT algorithm uses shared knowledge about the environment, as well as other heuristic variables, to make a choice. The applicable choices may represent any number of things, including (but not limited to) environmental navigation, behaviors, and agent interaction choices.

Genetic Algorithms can provide a behavior or a set of characteristics represented by a genetic sequence. This may be passed on to child agents in a mutated form, to produce new behaviors or combinations of characteristics. Also, an agent may mutate its own genetic sequence to promote new or different behaviors. In most cases, genetic sequences are subject to a fitness function, which weeds out poor candidates for future evolution, and replicates (with and without mutation) those remaining candidates. The result is a population changing towards the trend of the fitness function. This may be represented with a system call made by the system after n iterations, whereby agents are created and destroyed as needed. Otherwise, an agent’s genetic sequence may be modified to establish a new behavior or set of attributes in that agent, thereby preserving the agent, simply changing its internal characteristics parametrically.

Stimulus responses perform the task of absorbing stimuli, categorizing them, and responding. This may be represented by one or more of the above algorithms. An agent may utilize a classification/stimulus-response algorithm to learn about its environment, and react accordingly to changing environmental stimuli. Categorization refers to a storage technique whereby generalizations can occur, allowing the agent to react appropriately to new situations.

4. THE PROTOCOL

4.1 Client Connections

Client elements connect with MRScript and the connections are placed in a holding queue until the system is ready to initialize and add the connections to the list of active connections. A connection is inactive if it cannot receive communications via agents. Connections are made active only when the system is in a stopped state. Our experiments with auto-restarting the system (this involves an automated stop-start procedure) has shown that clients that connect in the midst of a system iteration are successfully picked up during the next recycling of the system. The purpose of this is to ensure that client elements are never forced into an unsafe, unstable, or confusing mode since in many cases commands build on other commands, or within a sequence of commands have semantic meanings that is compromised by allowing arbitrary connection times.

Once a client connects with the system, it transmits its name and waits for commands from MRScript. Since each client element has a name, the script can issue commands to particular client elements, or groups of elements. The communication system utilized by the scripting engine is buffered to ensure that the iteration of the system does not suffer from non-responsive clients, and is fail-safe in the case that clients drop off in the midst of system operation. These features, as well as the ability to easily define new client element protocols for different types of clients, make the communication system robust and effective.

4.2 Commands and Controls

It was discovered in an earlier version of the scripting engine that fine-grained control of graphical and audio representations of agents on client elements caused a severe network clog; messages were being transmitted too often, and the format of the data was unwieldy for such purposes. A solution has been to separate the command data from the control (physics) data.

Command data is now a textual stream containing only command statements, which modify the state of client elements or agents represented on them. Control data is a representation of the physics data intrinsic to every agent in MRScript. For example, the agent's location and orientation are two vital elements that are updated frequently in the case of fast moving agents (flocking fish are an excellent example of agents having this property). Control data may also contain linear and angular velocity/acceleration, to allow client elements to simulate agent movement at a rate faster than the transmission of control data. Control data transmissions are not linked explicitly to the iterative nature of the scripting engine, and so their granularity can be controlled independent of per-iteration update times.

An agent representation on a client element may use any control data transmitters (control sources) as issued by the scripting engine. MRScript operates as an intrinsic control source for all agents defined in it. Typically, the agent representations will connect to MRScript for their control data. They may, however, also connect to external control sources such as tracked sensors, other simulation systems, etc, so long as those systems implement the control data protocol. This means that agent representations (such as the user's view on a graphics client element) can be directly controlled (in terms of location and orientation) by tracked sensors mounted on a head-mounted display, for example.

A playful use of associating a tracking device with a model might be if we wished to place a fish head on one of the participants. The midpoint between the eyes of the fish model could be associated with the position of the tracker mounted on the HMD (or it might be placed slightly in front and below to align just in front of the user’s eyes). As the user moves his or her head, the fish model would move in unison. If the model is done with one-sided polygons, the user is unaware of the joke, but all other users see the fish head instead of the human head. The fish nods, shakes and moves with its tracked user.

Other examples could include placing a virtual object in a tracked moving vehicle or placing a futuristic weapon model on top of a tracked replica of a weapon being carried by the user.

5. QUICK OVERVIEW OF CLIENTS

The graphics and audio clients of our MR System employ a peer-to-peer strategy as regards their relation to MRScript. By this we mean that each agent managed by the scripting system that affects the visual presentation, e.g., those having a visual representation, whether virtual or real, has a peer in the graphics engine. Similarly, those with aural properties, e.g., virtual people having walking sounds when they move, have audio engine peers. These relationships are established via commands such as can be seen in the init script for the Rover object (Figure 2) used in our MR Sea Creatures experience (Figure 3).

The agent tag specifies the agent’s name, Rover, and the fact that it is model, which means it has associated geometry. The init tag means that these actions will occur at the start of the experience. The gfxcommands are sent to graphics engines; the audcommands are sent to audio engines.

The first graphics command tells the system agent in the graphics engine to make the Rover character. The graphics Rover character is then told to set its model according to the file “MRSC_rov.rov.” We are assuming that the location of this file can be found because a previous script for the MRScript system agent already instructed the graphics engine to create a world object and to load a package that identifies the location of assets, such as this model, that will be used to visualize objects in this world. The world object is then instructed to add the Rover agent to itself. Finally, the world is told to show the Rover. The hold=”n” parameter means that this is an asynchronous command; the script engine does not wait for a report back that the Rover has been shown. Other commands, such as creating the peer agent, are synchronous.

```
<agent name="Rover" type="model">
 <init>
  <!--create Rover-->
  <gfxcommand>system make {@SELF} as
      character</gfxcommand>
  <gfxcommand>{@SELF} set character
      MRSC_rov.rov</gfxcommand>
  <gfxcommand>world add agent
      {@SELF}</gfxcommand>
  <gfxcommand hold="n">world show
      {@SELF}</gfxcommand>

  <audcommand>system make {@SELF} as
      character</audcommand>
  <audcommand>Surround show
      {@SELF}</audcommand>
  <audcommand hold="n">{@SELF} loop
      MRSC_audio.rov</audcommand>
 </init>
</agent>
```

Figure 2. Init script for Rover agent

The audio commands are similar, although not typically as complex. The interesting thing here is that the Rover audio peer is a “surround” sound and that it has an associated sound file “MRSC_audio.rov” that loops until instructed to stop. By associating this sound with the Rover, its position is bound to that of the Rover. Achieving this positional audio is a major characteristic of our system. We do so by expecting the user to tell us where the speakers are located. This placement is used as a constraint for determining how to balance the sounds between speakers. With this scheme, we can deliver multi-tiered audio that users can perceive as moving vertically as well as horizontally.

6. MR SEA CREATURE EXPERIENCE

The MR Sea Creatures experience was installed at the Orlando Science Center from October 4-23, 2004. During this time,it ran from 10:00AM to 8:00PM, with none of our staff on-site. Start-up was done by OSC staff members merely by booting up the three machines we provided. Shutdown required then to run a single script on one system.

These machines all had 2GHz P4’s with 512MB of main memory. One was a single processor Win/XP system that ran the audio engine. The second and the third were dual processor Xeon’s Slackware 10.0 Linux systems; one ran the script engine, the HUD and a physics engine; the other ran the graphics engine with video capture coming from a mounted digital camera, along with a mouse server.

Figure 3 shows the view of the center from the MR Dome. The structure, redeployed from an exhibit we built for SIGGRAPH 2003, covers up the computers and some of the speakers (others are mounted hanging from the ceiling behind the users). In this figure, you can see several children in the museum (the real scene is captured by a camera mounted on the other side of the dome enclosure). Virtual objects include a Rover (2/3 up middle of image), an elasmosaur (long lizard-like creature on mid right side) and some smaller Cretaceous period sea life. As we previously captured the static objects in the real area, we use “occlusion models” of to partially occlude virtual objects that pass behind real objects like support poles and display cases.

The console was designed specifically for this experience. It provides a HUD (Heads Up View) of the world from the perspective of the Rover, whose position you control with a track ball and an up/down button. This display also shows a radar screen that indicates the position of objects of interest around you. When you get close enough to one of these, the HUD view is replaced by an information panel about the discovered object.

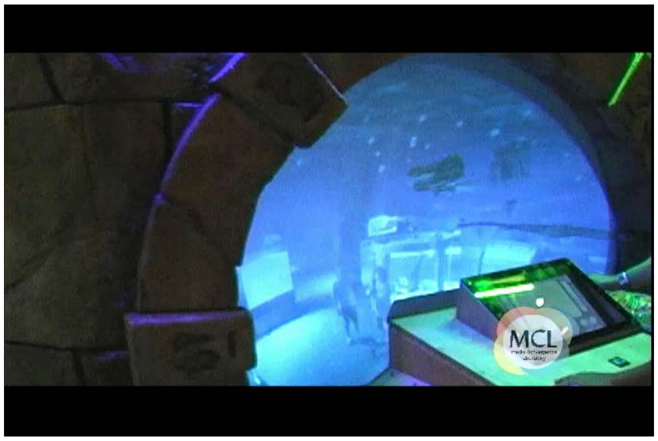

Figure 3. Cretaceous life at Orlando Science Center.

Figure 4 shows the back of the dome from a perspective within the museum exhibit. The camera that captures the real assets of the museum is mounted at the top middle of the back side, out of harm’s way.

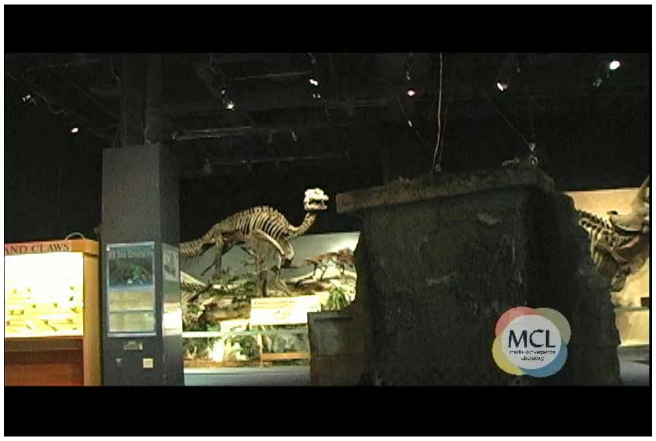

Figure 4. MR Dome from back side.

6.1 Successes and Failures

The software system ran cleanly, with a few camera driver glitches that resolved with a restart. The merging of real and virtual was a thrill for children who often became the objects being captured in their friends' experiences, once they realized what was happening. The interface worked terrifically with gamers and children in the middle school years (12-14). Younger children (4-11) tended to consume the experience; adults were confused by the interface unless taught by their children. Avid gamers teamed up (one controlled the track ball and the other the up/down buttons) in order to be successful at finding all objects of interest. The inexpensive flat screen we used for the console was a poor choice as it did not allow observers to see those details of the experience.

7. FUTURE WORK

We have recently added the concept of Auxiliary Physics Engines (APE). These engines operate through the network and can be dynamically added. The original motivation for these auxiliary engines was to solve the problem of calculating trajectories and then determining what objects are intersected by a given trajectory. Such a computation is clearly needed in experiences where weapons are fired, but it also needed in experiences where objects are selected by pointing.

In our previous versions of the MR system, we delegated trajectory computation to the graphics engine where this can be done very efficiently. We then had the story engine sort the intersected objects by distance from the selection device (e.g., gun). The story engine could then determine the semantics of an object being selected, and use the order of intersection to handle such things as bullets hitting walls before encountering vulnerable objects. Our first thought with our new design was to move this trajectory computation onto the scripting engine. However, since MRScript is written in Java and is already handling script interpretation and all experience oversight, we encountered unacceptable delays.

In the interest of flexibility and speed, we designed the notion of APEs and specified a common interface that supports trajectory computation, path planning (with optional collision detection), line-of-sight calculation and other, as yet undetermined, model-based services. In our first use of an APE, we employed the graphics engine as a provider of trajectory services; in this case the operation matches that of our older implementations.

Our goal now is to expand our experiments to employ dedicated APE service providers (computational workhorses). Each APE may provide a subset of basic services and some additional set of advanced features. APEs must support reflection (a method that reports the list of all services provided by this particular APE), so a potential user, e.g., agent, can determine if a given APE meets its requirements. APEs must also respond to a request to determine their current workloads. This can be used to do load balancing for complex large MR worlds.

Beyond the specific extension of MRScript with APEs and new AI plug-ins, our primary future efforts are focused on creating an "authoring by example" user interface [2][6]. In general, we wish to support the notion of an MR Backlot, with all sorts of reusable virtual assets and model behaviors that can be easily accessed, understood and employed in the authoring of new experiences.

ACKNOWLEDMENTS

This research was conducted within the Mixed Reality Audio research test bed at the University Central Florida's Media Convergence Laboratory, a partnership of the College of Arts and Sciences, School of Computer Science, and the Institute for Simulation and Training. The research is in participation with the Research in Augmented and Virtual Environments (RAVES) supported by the Naval Research Laboratory (NRL) VR LAB. It is also supported by the U.S. Army's Science and Technology Objective (STO) Embedded Training for Dismounted Soldier (ETDS) at the Research, Development and Engineering Command (RDECOM). Collaborative research and an ongoing dialogue are taking place with the Army Research Institute (ARI) at the Science and Technology Testing Center (STTC) in Orlando, Florida. Finally, special thanks are due to Nathan Selikoff for his being the first user / lab rat of this engine and to the Mixed Reality Laboratory, Canon Inc., for their generous support and technical assistance.

REFERENCES

[1] AMIRE Consortium. 2004. *Amire - Authoring Mixed Reality*. [Online]. Available: http://www.amire.net/ [November 18, 2004].

[2] Cypher, A. 1993. *Watch What I Do*. MIT Press, Cambridge, MA.

[3] Haringer, M. and Regenbrecht H. T. 2002. "A Pragmatic Approach to Augmented Reality Authoring." In *Proceedings of the First IEEE International Augmented Reality Toolkit Workshop* (Darmstadt, Germany, Dec. 29). IEEE, Piscataway, NJ, 237-245.

[4] Hughes, C. E.; Stapleton, C. B.; Micikevicius, P.; Hughes, D. E.; Malo, S.; O'Connor, M. 2004. "Mixed Fantasy: An Integrated System for Delivering MR Experiences." In *Proc. of VR Usability Workshop* (Nottingham, England, Jan. 22-23). [Online]. Available: http://www.view.iao.fraunhofer.de/Proceedings/ [November 18, 2004].

[5] Hughes, C. E.; Smith, E.; Stapleton, C. B.; Hughes, D. E. 2004. "Augmenting Museum Experiences with Mixed Reality." In *Proc. of Knowledge Sharing and Collaborative Engineering* (St. Thomas, US Virgin Islands, Nov. 22-24), ACTA Press, Calgary, Canada.

[6] Lieberman, H. 2001. *Your Wish is My Command*. Morgan Kaufmann, San Francisco, CA.

[7] Walt Disney Imagineering. 2004. *Panda3D*. Available [Online]. Available: http://www.etc.cmu.edu/panda3d/ [November 18, 2004].

[8] Zauner, J.; Haller, M.; Brandl, A.; Hartmann, W. 2003. "Authoring of a Mixed Reality Assembly Instructor for Hierarchical Structures." In *2nd International Symposium on Mixed and Augmented Reality* (Tokyo, Japan, Oct. 7-10), IEEE, Piscataway, NJ, 237-246.