Technology and Tools for the Creation of Reusable Visualization and Simulation Content in E-Learning Systems

Stephen H. Lane, Ph.D., Vincent Thomasino
soVoz, Inc.
Princeton, NJ
lane@soVoz.com, thomasino@soVoz.com

Bill Pike
U.S. Army RDECOM-STC
Orlando, FL
bill.pike@us.army.mil

ABSTRACT

Use of 3D visualization and simulation (VizSim) content can be a very effective learning tool for illustrating complex concepts, developing physical intuition and practicing "learning by doing" in a broad range of subject areas. Unfortunately, there is still a lack of cost-effective authoring tools allowing course instructors to seamlessly integrate and reuse such VizSim content in today's training and education applications. This paper describes a VizSim component framework, runtime and authoring tool that enables the creation of reusable VizSim content for Advanced Distributed Learning (ADL) environments that is conformant with the Sharable Content Object Reference Model (SCORM). The intent of the design is to allow course developers and instructional designers to easily create interactive VizSim application content using commercial off-the-shelf (COTS) graphical models, animations and behaviors. The VizSim content produced using the component-based approach presented is highly reusable and has application across a wide range of industries, including: education, training, industrial design, maintenance, entertainment and information technology.

ABOUT THE AUTHORS

Stephen H. Lane. Dr. Lane has been involved with the design, development and commercialization of state-of-the-art software technology for use in Interactive Entertainment, Virtual Reality, Computer Animation, and Distributed Training and Simulation applications for the last 15 years. Dr. Lane's responsibilities have ranged from basic technology development to computer hardware and software design, project management and business development in a variety of R&D projects with DARPA, the National Science Foundation, the Franklin Institute, the David Sarnoff Research Center, Princeton University and the US Army, and commercial development projects with Hasbro, AT&T, Microsoft, Disney, Intel, and others. He also has authored or co-authored 31 technical conference and journal publications. In 2001, Dr. Lane joined the faculty of the School of Engineering and Applied Science at the University of Pennsylvania as an Adjunct Professor in the Department of Computer and Information Science. Dr. Lane received the B.S. Degree in Mechanical and Aerospace Engineering from ***Cornell University*** in 1980, the M.S. Degree in Systems Engineering from ***UCLA*** in 1982, and M.A. and Ph.D. Degrees in Mechanical and Aerospace Engineering from ***Princeton University*** in 1988.

Vincent Thomasino. Mr. Thomasino is a seasoned manager and software engineer in the areas of computer graphics and distributed systems. Prior to joining soVoz, Mr. Thomasino worked at Eastman Kodak where he was responsible for the design and development of an award winning web-based workflow and imaging system. Before taking the position at Kodak, Mr. Thomasino developed a web-based document management system at Bell Atlantic that became the company standard for ISO9000 documentation. Mr. Thomasino studied at the ***Johns Hopkins University*** where he earned Bachelor's Degrees in both Computer Science and Political Science in 1996.

Bill Pike. Mr. Pike is the Lead Principal Investigator for Advanced Learning Environments at the US Army Research, Development and Engineering Command, Simulation Technology Center in Orlando, FL. He has led research projects in advanced distributed learning technologies such as intelligent tutoring systems, game engine-based simulation, and handheld training platforms. Mr. Pike earned his Master's Degree in Computer Engineering from the University of Central Florida. He holds a Bachelor's Degree in Systems Science from the University of West Florida and is currently working toward a Ph.D. in Modeling and Simulations at the University of Central Florida. He is also a Naval Reserve officer, currently assigned to US Special Operations Command at MacDill Air Force Base, FL.

Technology and Tools for the Creation of Reusable Visualization and Simulation Content in E-Learning Systems

Stephen H. Lane, Ph.D., Vincent Thomasino
soVoz, Inc.
Princeton, NJ
lane@soVoz.com, thomasino@soVoz.com

Bill Pike
U.S. Army RDECOM- STC
Orlando, FL
bill.pike@us.army.mil

INTRODUCTION

The Advanced Distributed Learning (ADL) initiative was launched to "ensure access to high-quality education and training materials that can be tailored to individual learner needs and made available whenever and wherever they are required (ADL Colab, 2003)." This initiative was designed to accelerate large-scale development of cost-effective dynamic learning software that meets the education and training needs of the military and the nation's workforce in the 21st century.

The ADL Co-Lab's Shareable Content Object Reference Model (SCORM) (Dodds, 2001), is a standardized technical framework that promotes the creation of dynamic learning content in the form of reusable components. Content creators use these components to author new forms of e-learning content while content consumers use industry-standard "runtime" systems (e.g. Learning Management Systems) to find and interact with the content constructed with them. These components are stored in a distributed network of SCORM-compliant repositories that are accessible through PCs, telephones and portable devices.

Use of Visualization and Simulation in Online Learning Systems

Use of visualization and simulation (VizSim) in ADL applications can be very advantageous for promoting learning comprehension and retention. For example, a 3D simulation of ground vehicles could enable technicians to practice various maintenance, assembly and repair procedures in a virtual environment or be used to train soldiers on operationally-correct strategies, tactics and rules of engagement in a game-like setting. Whether the goal is primary or secondary education; equipment assembly, maintenance or repair; the training of corporate personnel; or tactical decision making; VizSim enables students to "learn by doing" as well as easily explore concepts in strategic thinking, competitive tactics, and collaboration. In addition, the "game-like" nature of the interactive learning experience can lead to repeated usage and help encourage otherwise unmotivated or disinterested learners (Sawyer, 2002; Ahdell & Andersen, 2001; Pike, 2002).

Need for Cost-Effective VizSim Authoring Tools

The production of robust VizSim content using existing tools and techniques is prohibitively expensive in terms of both time and money. For example, the average video game takes two years and three to five million dollars to produce (Gamestate, 2003). As a result, use of 3D Vizsim content in online training and education applications is rather limited. A majority of the high cost originates from five sources: intellectual barriers to entry, bottlenecks caused by technical dependencies and communication problems between team members, supplemental R&D to support and maintain the production pipeline, failure to exploit content and code assets already developed, and a lack of a means to quickly and inexpensively validate design ideas.

This paper outlines a framework, runtime and supporting authoring tools for the structured design of SCORM conformant VizSim content based on reusable components. The toolset affords significant time and monetary savings by enhancing the effectiveness of existing production pipelines, thereby making the construction of VizSim content an option for any sized organization wishing to incorporate interactive visualization and simulation capabilities into their training and education applications.

HISTORY OF VIZSIM AND COMPONENTS

Component Technology

Component technologies allow for black box reuse of code. Technologies like COM, CORBA, EJB, and JavaBeans (Emmerich & Kaveh, 2001) have been in wide spread use for several years and promise to play an increasing role in application development. Application of component technologies to VizSim, however, has only just recently been attempted (Dorner & Grimm, 2001). Current approaches can be loosely defined as belonging to one of two categories: *Imperative* or *Declarative* (Mitchell, 1996).

Imperative approaches utilize an imperative programming language (e.g. C++, Java, etc.) to express the details of component operation where as declarative approaches employ a declarative programming language (e.g. HTML) to describe the components. In an imperative program the author must tell the system exactly **HOW** to solve the problem. Programs written in imperative languages are generally harder to write, debug, and maintain compared to those written in declarative languages. Programs written in imperative languages are also generally larger in terms of code size and run faster compared to programs written in declarative languages.

In a declarative program the programmer tells the computer **WHAT** the problem is and the system works out how to solve it. The programming model in declarative languages is based on stating the relationship between inputs and outputs. The actual solution adopted is left to the runtime system. A declarative program can be viewed as a high level specification. Declarative programs are shorter, supposedly easier to write, debug, and maintain. Declarative programs are also generally slower than imperative programs in terms of execution speed.

Example. An analogy is useful to clarify the difference between the two approaches. Suppose you are instructing a robot soldier to fire a gun at a target:

> ***Declarative Description****: Shoot Target.*
>
> ***Imperative Description****: Move gun to the left 5 degrees, raise gun 10 degrees, pull trigger…*

In the Declarative description it is easy to specify instructions the robot is to follow, but it requires a very sophisticated and intelligent system to carry them out. The Imperative case requires a much simpler system, but only authors with detailed task and system knowledge will know how to specify what the robot should do.

Imperative Approaches

Imperative approaches typically try to apply existing component technologies to VizSim. A number of frameworks have used this approach including Bamboo, i4D, Jamal, NPSNET-V and 3D Beans (Dorner & Grimm, 2001). Representative of the imperative approach is the solution called 3D Beans. The 3D Beans specification mixes JavaBeans with Java3D to produce a series of components that can be used in a variety of interaction metaphors to construct a 3D scene. The 3D Beans solution provides a tool called the Bean Box Editor that allows the assembly of components with minimal programming. Although imperative representations are typically more expressive and efficient at runtime, they suffer in terms of ease of use and accessibility by a non-expert.

Declarative Approaches

Declarative approaches on the other hand, are typified by the use of markup languages to produce structured documents that describe a component's interface, implementation, configuration, and assembly. Examples of this style of solution are 3DML, CONTIGRA, VRML, and X3D. Representative of the declarative approach is X3D (X3D Website, 2003). X3D is an XML-based file format for describing interactive 3D objects and worlds. The fundamental structure of X3D is the scene graph, which describes the visible and behavioral elements and their relationship to one another. X3D also specifies a runtime environment, which maintains the current state of the scene graph, renders the scene as needed, receives input from a variety of sources, and performs changes to the scene graph in response to instructions from the behavioral system. Although declarative representations are typically more accessible to the lay person, they suffer in terms of runtime efficiency and expressiveness.

Hybrid Approach to the Development of VizSim Components

Each of the above approaches to VizSim Component development has a number of drawbacks that limit their wide spread adoption. An analysis of these problems has led the authors to approach the task of designing a VizSim component framework based on the following principles.

- ***First, a non-expert should be able to configure and reuse VizSim content without referring to its creator.*** Past imperative and declarative approaches have required programming or artistic expertise to construct, configure, and/or reuse content. It is acknowledged that eliminating the programmer or artist from the production pipeline is impossible; however, it is possible to relegate their expertise to the construction phase. The present design approach addresses this problem by wrapping proprietary imperative representations of appearance and behavior with a simple declarative wrapper capable of modification from a graphical user interface. The accessibility of the declarative representation allows the lay person to configure and reuse imperative content without the expertise necessary to construct it. The present design approach utilizes a hybrid approach that utilizes

both declarative and imperative techniques where they provide the most flexibility.

- ***Second, extensibility must be balanced against accessibility.*** Previous solutions have provided for extensibility at the expense of accessibility. The frequency of content reuse is often determined by the accessibility of its representation. It is crucial that the component system be highly abstracted and consists of easy to recognize types whose use is explicitly defined. Too much extensibility creates a morass that non-experts may be unable or unwilling to wade through. Too much accessibility creates a limited closed system that hinders expression. The present design approach strikes a balance by not only providing a set of rigidly defined types, but also a mechanism for adding and changing types should the need arise.
- ***Third, content developers need multiple evolving options for representing appearance and behavior.*** Prior technologies have been tied to one format for appearance and/or behavior. Some solutions have reinvented the wheel by creating their own proprietary formats for representing appearance and behavior. This approach is usually self defeating as technology and developer preferences are changing so rapidly that it makes it almost impossible to stay relevant. The present design acknowledges this problem and has developed a component framework that can accommodate a variety of different appearance and behavior formats as they evolve over time.
- ***Fourth, the framework should focus the developer on developing content not infrastructure.*** Past solutions have required content developers to engage in infrastructure development to make their VizSim content run efficiently. This infrastructure development was often done at the expense of content development. Some of the previously mentioned solutions have added management infrastructure they lacked as an afterthought; however, this has resulted in numerous backward compatibility problems with previously developed solutions. The component system described in this paper frees the developer from creating run-time component infrastructure and focuses him exclusively on content development.
- ***Fifth, an encoding is only as good as its tool support.*** It doesn't matter how simple the encoding is, non-experts will always prefer to author with a GUI. Therefore, an encoding must be designed from the ground up to support use in a visual tool. Extensions must be provided whereby a component creator can annotate the various pieces of their component so that it may be inspected by a tool. Early component frameworks neglected extensions for tool support and consequently impeded the reuse of their content. The component framework outlined in this paper has an extensible means of adding Meta-data to a component to aid in tool integration.
- ***Finally, the execution model must be sufficiently rich to describe the spectrum of simulation paradigms.*** Previous frameworks have provided nonexistent or simplistic execution models that were inadequate for robust simulation. Moreover, those execution models were sometimes tightly coupled with a scene graph thereby making independent information processing awkward if not impossible. The robust execution model described in this paper makes provisions for layering, blending and sequencing of behaviors in both series and parallel, independent of the simulation's visual representation.

A PROTOTYPE VIZSIM COMPONENT SYSTEM

The sections that follow present a framework, runtime, and supporting authoring tools for the structured design of VizSim content based on reusable components. The resulting system design is compatible with existing Advanced Distributed Learning (ADL) environments and capable of producing content that is conformant with the Sharable Content Object Reference Model (SCORM).

VizSim Component Framework

The component framework shown in Fig. 1 provides a format neutral means of describing the relationships between disparate formats for appearance and behavior. VizSim components act as wrappers for imported media assets (models, code, animation data, sounds, etc.) which can be set in relationship to one another to describe complex visualizations or simulations.

Encoding

Components can be encoded in either an object oriented script language like Python or an object oriented system language such as C++. Script languages provide a number of advantages for those looking to reduce the cost of development such as platform independence and rapid application development. System languages can be used to

maximize performance by producing highly optimized platform specific code.

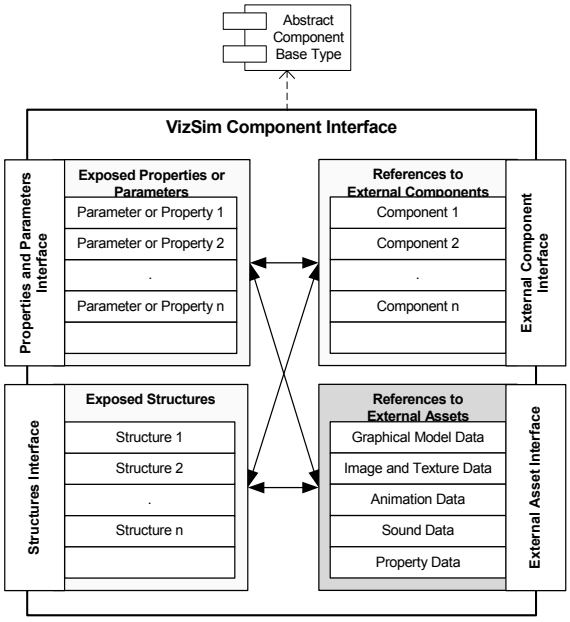

Figure 1. VizSim Component Framework

In the current implementation of the Component Runtime, components can be written in either Python for platform independence or C++ for exceptional performance. A unique feature of the Component Framework is the functional parity between script languages and system languages. End users can build components using either Python or C++ without making sacrifices in terms of expressivity. Script languages are not second class citizens in the Component Framework.

Providing the option to use different binary implementations is important for encouraging adoption of the framework. It is the authors' judgment that the currently supported languages encompass the largest body of legacy code and programming expertise available today, however, support for other languages such as Java, C#, Ruby, Perl, etc. could be added later as the need arises.

Composition Features

The feature set of the Component Framework focuses on providing rich metaphors for component construction and reuse.

Inheritance. The framework will support the idea of reuse through inheritance. Through this concept a new type can be declared that is an extension of an existing type. This new type is said to derive from the existing type and is called a "derived" type. The original type is the new type's base type. For example, a base Actor component can be defined to represent a biped. All bipeds have a head, two arms, a torso, and two legs. That biped component can be specialized into a soldier component with the addition of features such as a gun or backpack. Such derivation provides a way of expressing an "is a" relationship. The component developer need not specify the features of the base type in a derived type. Depending on the underlying implementation languages, inheritance in the framework need not be limited to single inheritance meaning it is possible to derive a new type from multiple base types. Python components can seamlessly extend C++ components though single or multiple inheritance with full bidirectional support for polymorphism.

Reflection. Components are reflective or introspective because they contain sufficient meta-data to fully describe the data they represent. This means that given the address of an component, it is possible to ask enough questions of the Runtime to fully "decode" the data the component at that address represents. Reflectivity allows the runtime to gather internal information about a component for use within the tool.

Mutation. The framework exposes the ability to create new component descriptions at run time and to add new properties to existing components. One common use of mutation is when providing a feature, like physics or sound, that needs extra data (like mass or friction) added to existing components such as geometric descriptions. With the Framework's mutable reflective component model, this is easy and allows existing components to be annotated with special data at will.

Containment. Finally, components can be combined at runtime through containment to yield new components with different properties. For example, behaviors can contain other behaviors by designating them as sub behaviors. A behavior and its sub behaviors represent a logical unit that can be reused as a part of a larger behavior graph. For example a "Fire Prone" behavior can be created by combining a behavior that makes a soldier lay prone with a behavior that makes the soldier discharge his weapon. Libraries of complex components can therefore be built from primitive ones.

Component Anatomy

Components in the present Framework are created by deriving them from abstract component types. Abstract component types cannot be used in their own right as

they are templates for construction of concrete components. Abstract component types are typically written in imperative programming languages like C++ or Python and can be distributed with the runtime or individually as needed. Each of the abstract component base types has different runtime semantics that determine the role the component will play in a VizSim application. The current Component Framework specifies five abstract component base types:

- **Production** - Represents a collection of scenes which when combined form a simulation.
- **Scene** – Represents a 3D space where an author constructs a portion of a simulation.
- **Actor** - Represents the state, geometry, appearance, kinematics, dynamics, etc. of an entity in a simulation.
- **Behavior**[1] – Represents parameterized operations on the state of an Actor in a simulation.
- **Event** – Represents parameterized tests and notifications for something of interest in a simulation.

The framework also provides several derived abstract base types to aid in simulation construction, including:

Deriving from **Actor**:

- ***Camera*** - Represents as view point into a scene.
- ***Light*** – Provides illumination in a scene.
- ***Sound*** – Represent 3D audio in a scene.
- ***Text*** - Represents 3D text in a scene.
- ***Curve*** – Represents one-dimensional curves in 3D space.
- ***Timer*** – Generates events at specified intervals.
- ***Group*** - Represents collections of actors that can be combined and manipulated as a logical unit.

[1] This document uses the term "behavior" to mean interactive movements and actions that have varying levels of complexity. For example, a behavior can be as simple as a curve that specifies the goal location of an Actor's hand as a function of time or as complex as the complete set of strategies and tactics that coordinate the movements and actions of a Dismounted Infantry Fire team.

Deriving from **Behavior**:

- ***Meta*** – The top level behavior of an actor, production, or scene.
- ***Sequencer*** – Provides goal-direct transitioning between behaviors states.

Deriving from **Event**:

- ***Behavior Status*** - Signals the state of a behavior in its execution cycle.
- ***Proximity*** – Signals the intersection of two bounding areas.
- ***Temporal*** – Signals an expired duration or regular interval.
- ***Keyboard*** – Signals the manipulation of the keyboard.
- ***Mouse*** – Signals the manipulation of the mouse.
- ***Joystick*** – Signals the manipulation of the joystick.

The collection above of abstract base types can be used to build a library of VizSim components for the construction of complex VizSim content.

Entity Types vs. Operation Types

The abstract component base types described above can be divided coarsely into two categories: Entity and Operation. Entity types represent state in the system where as Operation types represent manipulations or tests of system state. The types Production, Scene, Actor and any types derived from them are considered Entity types. Entity types can maintain state throughout their lifetimes and are used as namespaces for storing information in a simulation. The types Behavior, Event and any types derived from them are considered Operation types. Operation types manipulate Entity types but maintain no persistent state of their own. The runtime can achieve great scalability by using various techniques that rely on Entity types being stateful and Operation types being stateless.

Meta-data and Tool Support

VizSim components are designed from the start to be used in a variety of different runtime and authoring environments. A number of features have been added to the encoding to aid in application integration. This section lists the specific Meta-Data extensions necessary for integration.

Context Sensitive Help. Each component has a Meta-data section that can convey general information to the user and can be exposed in a tool for inspection.

Dynamic User Interface. A tool can inspect the declaration of a component to construct a means of manipulating it. Each component declaration provides all the information necessary to build a dynamic user interface capable of manipulating it.

SCORM Support. Meta-data encoded in a component declaration or instance can be utilized to automatically generate SCORM compliant Meta-data descriptions. See section on Meta-data generation under Component Authoring for details.

Security

e-Learning solutions often involve many different participants and the VizSim e-Learning solutions of tomorrow will be no exception. Each participant will have different roles and security requirements with respect to a simulation. (i.e. student, instructor, administrator, etc.) The Component Framework addresses this situation by providing a robust security mechanism as a part of its design. The component framework provides role-based security as an automatic service that enables one to administratively construct and enforce an access control policy for a production.

Serialization

To achieve scalability components are cleverly juggled in and out of memory by the runtime. Components therefore need to be able to persist themselves at any time. The framework provides facilities to aid in serializing a component's internal state when required by the runtime. Components are serialized as XML (XML Website, 2003) documents and validated with the W3C's XSD (XSD Website, 2003). XML provides a number of advantages such as platform independence, standardization, interoperability, tool support and powerful transformation and processing models.

Registration

The Component Registry is a runtime database that contains information about the components composing a production. When a production loads, the runtime collects meta-data about all of its constituent components and prepares them for instantiation. The framework provides a mechanism for production authors to specify administrative meta-data that can be used to change the runtime behavior of a component between deployments.

VIZSIM RUNTIME

The VizSim Runtime shown in Fig. 2 consists of the **Component Manger**, the **Behavioral Operating System (BOS)**, and the **Display Engine**. In the discussed system, the Component Manager manages the lifecycle of a component, the BOS is responsible for the coordinated execution of behaviors, and the Display Engine is responsible for the multimedia presentation.

Component Manager

The Component Manager is one of three major subsystems in the runtime. It provides component lifecycle services to the Behavioral Operating System. The Component manager is designed to take the complexity out of writing VizSim components. It allows VizSim component authors to stay focused on writing application specific content as opposed to the infrastructure necessary to run it safely and efficiently. The Component Manager provides a secure scalable environment for component execution that can be configured for the needs of the target platform at deployment time. Major features of the VizSim Component Manager include:

Component Retrieval and Creation. When the Component Manager is instructed to load a Production, it must first verify that all of the components referenced by the production's accompanying manifest are installed locally. The manifest lists all of the component dependencies in a production and the location of the assets that compose them. The Component Manager inspects recursively all of the component declarations referenced in the manifest and installs any assets they reference locally. This facility frees the component creator from having to worry about component dependencies at runtime.

Registry Maintenance. The Component Registry is the runtime database used to keep track of loaded components. The registry maintains information about all the components involved in a production. The Component Manager uses this registry to fulfill component creation requests. It is the Component Manager's responsibility to keep the registry up to date as new components are added or removed from the Production. This feature frees component writers from having to think about mundane component distribution issues.

Security Management. The Component Manager also provides several security features that can be used to protect Productions. The automatic security services that the runtime offers—role-based security

and authentication—make it possible to leave all security-related functionality out of components. When the services are turned on and appropriately configured, the runtime will handle the details of enforcing the security policy specified.

Component Pooling. Component pooling is an automatic service provided by the Component Manager that enables component instances to be kept active in a pool, ready to be used by any client that requests the component. Very significant performance and scaling benefits can be achieved by reusing components in this manner.

Just-in-Time Activation. Just-in-Time (JIT) activation is an automatic service provided by the Component Manager to help a production use resources more efficiently, particularly when scaling up a production to involve large numbers of components. When a component is configured as JIT, the runtime can deactivate it while another component still holds an active reference to it. The next time a client references the component, the runtime reactivates the component transparently to the client, just *in time*.

Serialization. In order to support the resumption of a simulation from a saved point the Component Manager may interrupt the execution of a simulation at anytime and instruct all components to save their current state. Upon restart of the simulation the Component Manager reconstructs the simulation as it was at the time it was saved.

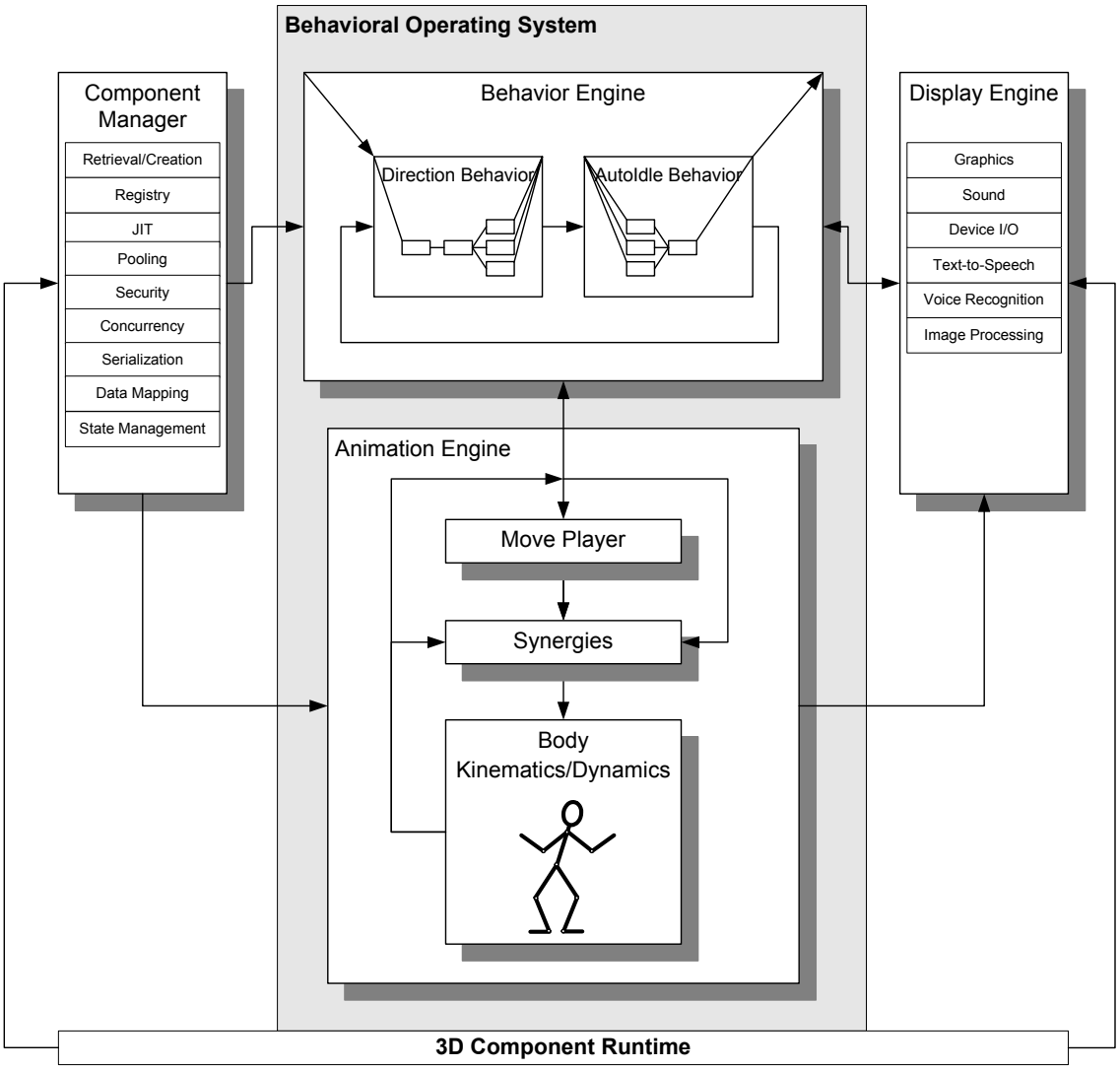

Figure 2. VizSim Component Runtime Architecture

Behavioral Operating System (BOS)

The Behavioral Operating System (BOS) is a platform-independent simulation engine for building interactive and dynamic VizSim content. The BOS implements a Sense, Control, Act simulation paradigm as its execution model.

Behavior Representation. Behaviors are represented in the BOS as hierarchical goal-directed parameterized finite state machines (Havel & Politi, 1998). At the lowest level, behaviors supply a wrapper that provides a list of starting events and ending events (for triggering actions on and off), a list of parameters (e.g. speed, direction, etc.) and an execution state (idle, active, done, etc). At the mid-level, behaviors can supply lower-level behaviors with goal values (such as the desired positions of the hands, head, feet and center-of-gravity as a function of time), set parameters, trigger subsequences on and off, and communicate with the various User Interface Subsystems. (Graphics, Sound, Device I/O, Text-to-Speech, Voice Recognition, and Image Processing)

Behavior Graph. Every actor has a special type of behavior called a MetaBehavior, which is the root behavior of the behavior graph. When the BOS executes an Actor's MetaBehavior, the actor comes to life. All other behaviors associated with an actor are considered children (i.e. sub-behaviors) of the Actor/MetaBehavior. Sub-behaviors are also hierarchical in that they can contain further sub-behaviors.

Events. Events are used to trigger or affect the sequence of actions an Actor performs during the execution of a behavior. Events can be generated based on the system clock, by system events, through program logic, or by external devices. There are three major types of events in the system.

- *Starting Event* - a Boolean expression that when evaluated to true causes the behavior to begin execution.
- *Ending Event* - a Boolean expression that when evaluated to true causes the behavior to terminate execution.
- *Temporal Events* - a list of temporal cues relative to the behavior's start time that when evaluated to true cause the behavior to send messages to the BOS event handler to trigger various actions such as voice and audio playback, graphical special effects and system commands such as mouse clicks, key presses and standard Windows messages.

Display Engine

The Display Engine is used to manage the low-level details of graphics rendering, sound generation, device IO processing, image recognition, text-to-speech, and voice recognition. The system was designed to work so that VizSim content can be developed and ported to a variety of different operating systems (Windows, Unix, etc.), media layers (DirectX, OpenGL, etc.) and hardware platforms. (PC, handheld devices and Game Consoles).

VIZSIM AUTHORING

In order to evaluate the efficacy of the Framework's design, an authoring tool was developed. The design requirements for the VizSim Authoring Tool were as follows:

- Provide intuitive interactive authoring capabilities leveraging behavioral animation techniques.
- Combined traditional linear and nonlinear approaches to animation.
- Enhance existing production pipelines and techniques so as to make them more efficient and cost effective.
- Establish workflows that eliminate dependencies between team members and allow team members to work in parallel.
- Allow VizSim content to be constructed incrementally using "off-the-shelf" components.
- Support reuse by allowing import/export to a SCORM-compliant Repository.
- Promote a "design, test, refine" paradigm throughout the development cycle.

Working as a Director

The objective of the prototype authoring tool was to enable a content expert to approach the job of creating interactive VizSim content from the perspective of a "Director" working within the following authoring paradigm:

All VizSim content is composed of reusable VizSim components. The author considers himself in the position of the "Director" and wants a particular scene acted out. He thinks first in terms of the goals he wants the VizSim Components (e.g. his actor) to achieve and the sequence of tasks that they must perform in order to achieve these goals. As he proceeds with the design he begins to select

VizSim components and invokes relevant goal-directed behaviors; e.g. move to the (x,y,z) location I am pointing to, pick up part P, attach the part to the object I have selected, etc. to implement the subtasks. The author assumes that the components can do what he asks of them and is prepared to instruct them with any portion of the task that they cannot accomplish by themselves. Eventually the tasks are broken down into simpler and more specific subtasks as he determines the ability or lack of ability of the VizSim components he uses. In the end he may have to specify exactly how the objects should move in order to accomplish the task he's instructed them to perform.

The approach described above leads the author to design simultaneously from the top down and bottom up to produce the desired result. The system also is designed to allow for refinement and trial execution of any portion of the scene as the design proceeds. This means that a VizSim component (e.g. actor) can be interactively tested once its geometry, joint hierarchy and behaviors have been defined. The goal-directed and parameterized nature of the FSM behaviors also allows the resulting task execution to be easily modified. As tasks are defined and implemented they can be repeatedly refined until satisfactory results are achieved. In addition, cues that have originally been defined as explicit functions of time can be easily exchanged for cues that are event driven. For example, instead of reaching out to grab a part at a specific instance in time, an Actor can be made to start reaching when a proximity detector determines it is within one foot of the part. This allows the task simulations to be generalized and reused in different contexts.

VizSim Authoring Tools

The graphical user interface (GUI) of the VizSim Authoring Tool prototype is shown below in Fig. 3 and consists of the following parts:

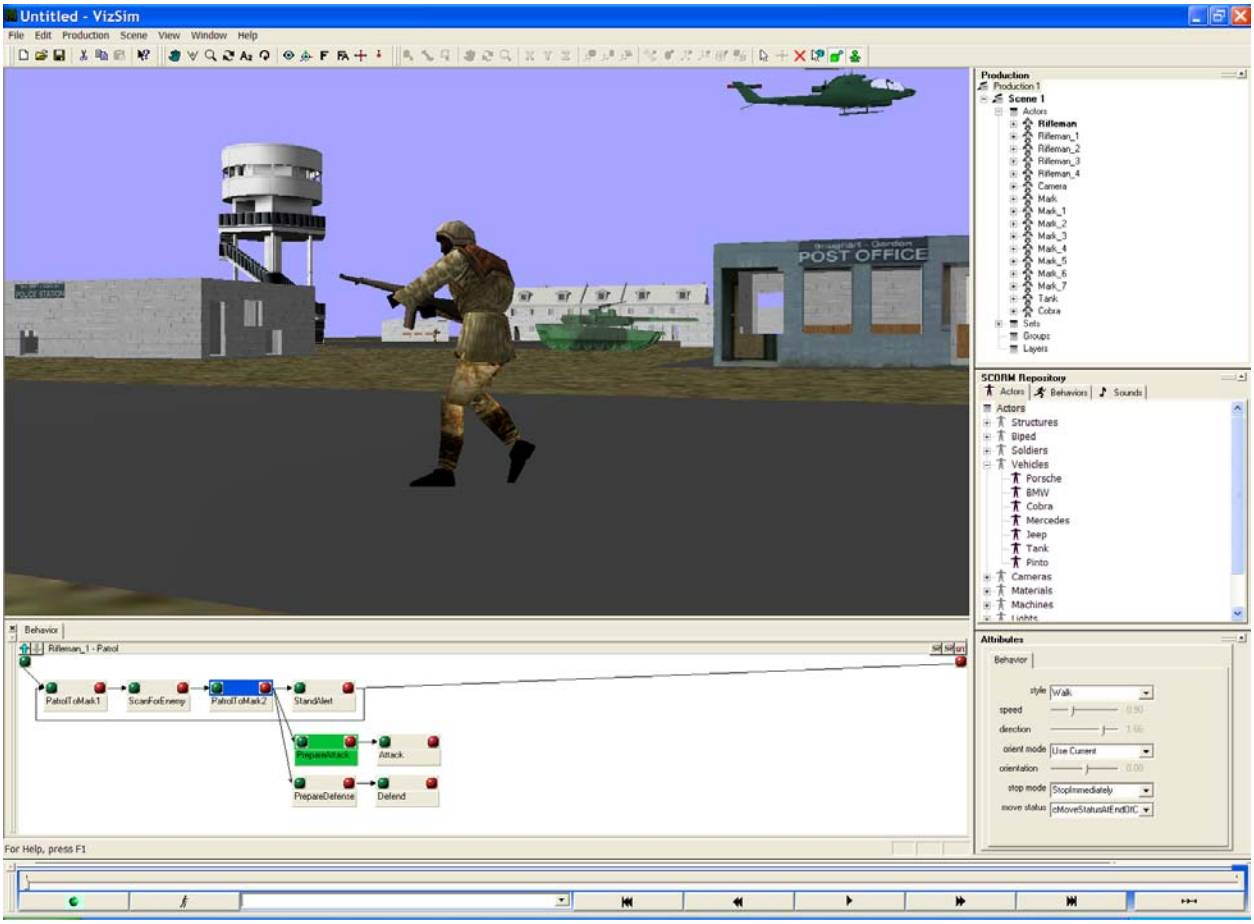

Figure 3. VizSim Authoring Tool Prototype

Production Window. The Production Window shown in Fig. 4 allows users to manage the contents of the current Production as a hierarchical tree structure. All components that have been inserted into a Production are visible from this window. Components are organized into collections under their respective containers. Selecting a component in the Production Window makes it the active component in the scene and thus subject to manipulation.

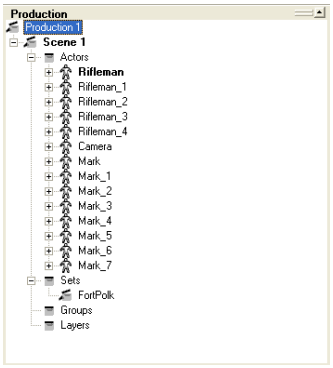

Figure 4. Production Window

Scene Window. The Scene Window shown in Fig. 5 displays a three dimensional rendering of the current scene. A Scene Window can selectively display a scene's entities and resources from a variety of different perspectives. The Scene Window offers an assortment of navigation tools that can be used to create new perspectives. A user may open multiple scene windows for different real-time views of the same scene. Actors in a scene can be manipulated directly from the Scene Window.

Figure 5. Scene Window

SCORM Repository Window. The SCORM Repository Window, shown in Fig. 6, allows the user to use predefined components to create Productions. Each component type is represented by a tab. An author drags components from the SCORM Repository Window to the Production Window to add them to the current Production. Meta-data within a component can be used to categorize components and make them easier to choose from.

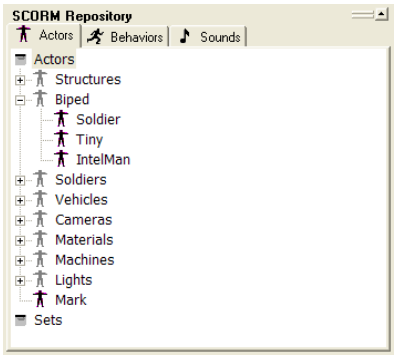

Figure 6. SCORM Repository Window

Attributes Window. The Attributes Window, shown below in Fig. 7, allows all exposed Actor properties and Event/Behavior parameters to be interactively adjusted at design and debug time. The user interface is constructed dynamically by inspecting a component's declaration and meta-data. Easy access to Actor properties and Behavior/Event parameters facilitates the test and refine approach to behavioral animation and VizSim Component development presented in this paper.

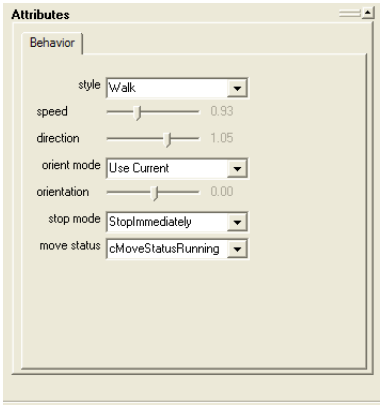

Figure 7. Attributes Window

Behavior Window. The Behavior Window shown in Fig. 8 provides an interface for defining the behavior of entities. Each entity in a production is represented by a tab. Within each tab a user defines a directed graph of behaviors that specify how the entity will act in certain situations. Behaviors are represented as blocks. Transitions between behaviors are represented as arrows entering the behavior from

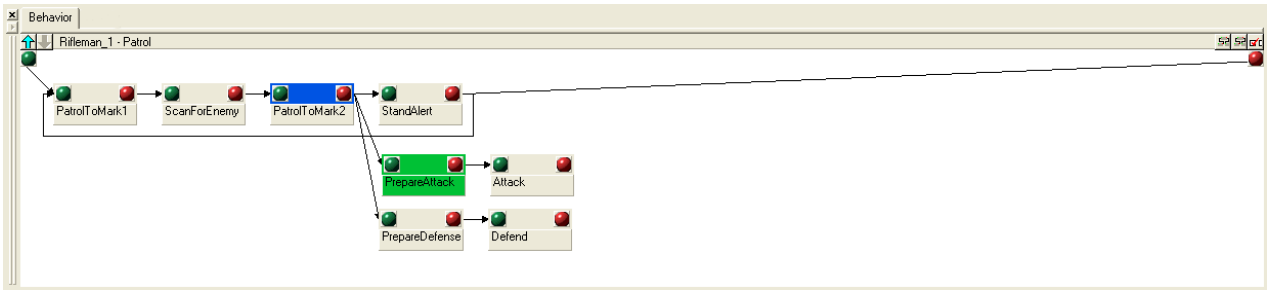

Figure 8. Behavior Window. The lines connecting pairs of behavior blocks represent behavior dependencies and the green and red buttons in the upper left and right hand corners the lists of Boolean starting and ending events that cause one behavior to terminate and another to begin.

the left and exit from the right. Behaviors are connected to one another by dragging a Transition from the end event of one behavior to the start event of another. A Transition is triggered when the starting events of the behavior tests true. Upon triggering a starting event behaviors cycle through a series of states: Idle, Active, and Done. The state of a behavior is denoted by changing its color in the view. Idle behaviors are colored gray, active behaviors are green, and done behaviors are red. Some behaviors will have sub-behaviors. Sub-behaviors are accessed by double clicking on their container behavior.

Recorder Window. The Recorder Window, show below in Figure 9, provides interactive control over behavior execution. Pressing the "play" button makes a selected behavior active. Pressing the "stop" button stops the selected behavior. If a behavior utilizes motion data the "advance to end", "advance to beginning", "forward", "rewind" "record" and "looping buttons can be used to control the playback of the animation.

Figure 9. Recorder Window

Authoring of VizSim content

Composition. VizSim content is created by following the procedure below:

1. Drag a new or pre-existing Production component from the SCORM Repository Window to the Production Window to begin working on a production.
2. Drag Scene components from the SCORM Repository Window to the Production Window to add scenes to the current production.
3. Drag and drop appropriate Actor components from the SCORM Repository Window into the Production Window.
4. Drag and drop appropriate behaviors from the SCORM Repository Window into the appropriate Actor's behavior graph in the Behavior Window.
5. Create direction by wiring-up the various behaviors in the Behavior Window, configuring their respective start and end events, assigning goals and setting parameter values.
6. Group behaviors, if and when appropriate, to create hierarchical Direction and to facilitate reuse through export to a component repository.
7. Repeat steps 3 through 6 for all the other actors in a scene and all scenes in a production.

SCORM Meta-data Generation. Meta-data encoded in a component can be utilized to automatically generate SCORM compliant Meta-data descriptions. The authoring tool uses this mapping to generate appropriate SCO and Asset Meta-Data when publishing a component to a SCORM repository.

Preview, Debugging and Publishing. The author can preview and debug a Production at any time during the VizSim content development process by selecting a particular behavior in the Behavior Window and clicking the "play" or "stop" button in the Recorder Window as appropriate. Parameters of a particular behavior also can be inspected and/or modified by selecting the behavior in the Behavior Window and viewing/changing the parameter values in the Attributes Window. Once the desired interactivity of the VizSim content has obtained, the production can be published as a SCO in a SCORM Repository, for viewing in a webpage or as a standalone application.

CONCLUSIONS

The ability to add VizSim to instructional content can make such content more intuitive, intelligible, and effective. Current methods for producing this type of content are time consuming and require the services of professional artists and programmers, making its use by individual course instructors cost prohibitive. The approach described in this paper can change this by allowing VizSim content to be integrated into ADL learning applications in a cost effective manner through a component framework with the following design goals and supporting features:

Design Goal	Features
Interoperability – The system shall be dissolvable into smaller pieces capable of being used in a variety of different applications.	• The component framework uses platform independent representations for components. (i.e. C++, Python, etc.) • The runtime can be hosted in a variety of different environments. i.e. Web Browser, Standalone Executable, etc.) • The authoring tool's GUI is designed modularly such that it may be integrated with other development environments.
Accessibility – The system shall employ metaphors and methodologies appropriate to the development audience.	• The framework can use industry standard representations (i.e. C++, Python, etc.) to describe components allowing them to be developed with existing tools and expertise. • Components consist of straightforward types with explicitly defined roles so that they may be understood and used by a non-expert. (e.g. Productions, Scenes, Actors, Behaviors, Events, etc.)
Reusability – The system shall produce VizSim content capable of being used again or repeatedly.	• The framework uses industry standard representations (i.e. C++, Python, etc.) to describe components allowing them to be developed with existing off-the-shelf content. • The framework is based on an open specification allowing a component to be developed with one vendor's tool and reused by another. • Components are SCORM compliant and capable of being used in existing ADL environments.
Extensibility – The system will be designed to evolve as requirements change.	• A component's description separates appearance and behavior allowing both to evolve independently. • The runtime interface is platform independent so that VizSim application content can be developed and ported to a variety of different operating systems (Windows, Unix, etc.), media layers (DirectX, OpenGL, etc.) and hardware platforms (PC, handheld devices and Game Consoles)
Affordability – The system will increase learning effectiveness while significantly decreasing development time and costs.	• The framework confines the need for professional programming and animation expertise to the component construction phase. • VizSim components are configurable and reusable by designers and domain experts without the assistance of highly-paid professional programmers and animators. • The framework provides reuse features that allow different parts of an application to be developed in parallel rather than sequentially. • The authoring approach reduces complexity by promoting a common development model for designers, artists, programmers, and instructors. • The tool provides an interface to component repositories to facilitate reuse of components across projects and teams. • The tool reduces the risk associated with developing complex simulations by encouraging the use of proven tools and design processes. • The tool increases reliability of simulation content by allowing simulation content to be authored using proven pre-existing components as basic building blocks.

ACKNOWLEDGEMENTS

This work was sponsored in part through an SBIR contract with the Office of Secretary of Defense.

REFERENCES

ADL Co-lab Website. (2003). ADL Overview, http://www.adlnet.org/index.cfm?fuseaction=abtadl

Ahdell, R. and Andresen, G. (2001). *Games and simulations in workplace e-learning*,, Master's Thesis, Norwegian University of Science and Technology, Dept. of Ind. Econ and Tech. Mgmt

Dodds, P. (2001). *Advanced Distributed Learning Sharable Content Object Reference Model (SCORM), Version 1.1*. Available via the Web at http://www.adlnet.org

Dorner, R. and Grimm P. (2001). Building 3D Applications with 3D Components and 3D Frameworks, *International Workshop on Structured Design of Virtual Environments and 3D-Components at the Web3D 2001 Conference*, Paderborn, Germany, February 19th. (http://www.c-lab.de/web3d/VE-Workshop/schedule.html)

Emmerich, W. and Kaveh, N. (2001). Component Technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA Component Model, In V. Gruhn (ed). *Proc. of the Joint 7th European Software Engineering Conference and 9th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, Vienna, Austria. ACM Press.

GameState Magazine (2003). Don't throw it all away, Spring, pp. 6-11.

Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, (with M. Politi), McGraw-Hill.

Mitchell, J. (1996). *Foundations of Programming Languages*, MIT Press, ISBN: 0262133210

Pike, B. (2002). Game-based Training Technology, *23rd Army Science Conf.*, Dec. 2-5, 2002, Orlando, FL.

Sawyer, B. (2002). *Serious Games: Improving Public Policy through Game-Based Learning and Simulation*, Foresight and Governance Project Woodrow Wilson International Center for Scholars, Publication 2002-1.

X3D Website. (2003). http://www.web3d.org/x3d/

XML Website. (2003). http://www.w3.org/XML.

XSD Website. (2003). http://www.w3.org/XML/Schema.