An Architecture For Domain-Independent Collaborative Virtual Environments

Ahmed BinSubaih
Steve Maddock
Daniela Romano
Department of Computer Science
Regent Court, 211 Portobello Street
Sheffield, S1 4DP, UK
Email: a.binsubaih, s.maddock, d.romano@sheff.ac.uk

ABSTRACT

The increase of computing power and its wide availability has raised interest in the use of Collaborative Virtual Environments for training purposes. Nevertheless, many of the currently developed training simulations are inflexible and support only a limited number of scenarios, often limited to a single domain. This work aims to investigate the challenges of separating the domain logic from the system using it. The work presented in this paper proposes to achieve the above by separating the architecture into three parts: domain knowledge, simulation environment and *events space* which links the previous two. The domain knowledge holds the scenario logic or behaviour which dictates how the scenario should run. The simulation environment is where participants meet and interact. The *events space* links the two parts by using the domain knowledge to control the scenario running in the simulation environment. By formulating our system in this manner we attempt to achieve the flexibility pursued and identify and tackle the challenges involved.

Keywords

Collaborative virtual environment (CVE), knowledge-base systems (KB), architecture, events

INTRODUCTION

A number of virtual environments (VEs) and collaborative virtual environments (CVEs) have been developed over the years, looking at such things as training firefighters (Romano 2001), police officers (Williams 1995) and navy personnel (Hamman et al 2001a), and the reconstruction of traffic accidents from textual reports (Akerberg et al 2003). The main issue with such systems is that they are domain dependent, which makes it difficult to reuse their simulation systems on different domains without extensive work. However, this is an expensive and time-consuming process. As identified in (Dachselt 2001) domain specificity in system development leads to highly inflexible applications.

Our work investigates how the disciplines of distributed environments and expert systems can be combined in an attempt to achieve the separation required which we believe leads to a domain-independent architecture. From distributed environments we investigate the suitability of events as a communication mechanism between the simulation environment and the *events space*. From expert systems we examine the applicability of using knowledge-base systems (KB) to store and inference the domain knowledge which is used to control the scenario.

One of the major contributions of this paper is introducing an *events space* which is an intermediary between the scenario logic and the simulation environment. Using such an intermediary should result in decoupling the scenario logic from the simulation environment which confers several advantages:

- The logic and the simulation environment can be modified entirely independently allowing iterative development cycles.
- The decoupling encourages encapsulation and other good object-oriented coding practices.
- It enables interoperability between distinct simulation environments, a practice promoted by the High Level Architecture (HLA) (Smith 1998). Although HLA promotes interoperability, its object management allows the subscription and discovering of remote objects thus violating the space decoupling[1]. Further concerns of the HLA are covered in (Davis and Moeller 1999)
- The three different parts (KB, *events space*, and simulation environment) can be individually tailored to the expertise and computer literacy of their users (domain experts, trainer, and trainees).

Furthermore, the *events space* attempts to achieve the following goals:

- Automatic generation and control of scenarios for training purposes.
- Use the ability of KB systems to provide explanation of solutions to guide trainees during simulation sessions.

The inherent distributed nature of a CVE and the existence of many different techniques for communication, such as the events mechanism with its decoupling ability (Drew et al, Eugster et al 2003), have encouraged us to attempt the separation on a collaborative virtual environment rather than on a VE.

In this paper, we first describe the related work, followed by a detailed presentation of the proposed architecture where we show the different structures and how they communicate to achieve the independence sought. Finally, we illustrate the first prototype developed by showing its ability to run two distinct scenarios: investigating the aftermath of a vehicle accident situation and virtual lecturing.

RELATED WORK

In this section we describe the methods used to structure simulation environments and categorize them with regards to the relationship between them and the scenario logic. Such categorization examines this relationship in terms of the communication approaches employed to control the scenario logic and the flow of the scenario logic from its source to the simulation environment.

We can consider the following different categories based on the relationship between the scenario logic and the simulation environment: applications, virtual development environments, commercial software environments, and KB systems.

Certain applications (Romano 2001) tend to embed the scenario logic inside the simulation environment code and most likely changes to the logic require recompiling the simulation system. Other applications (Akerberg 2003, Hamman et al 2001a, Williams 1995) attempt to give some ability to modify the logic without recompiling, thus eliminating the developers involvement in the scenario creation cycle. This is accomplished by providing ways for the domain experts or trainers to insert the logic into the simulation environment. However, these applications are usually very domain dependent.

[1] Space decoupling (Davis and Moeller 1999)

Some virtual development environments (Cruz-Neira et al 2002, Hawkes and Wray 1998, Shaw et al 1992, Tamberend 2003, Wang et al 1995) are built specifically for the use of developers producing virtual environments. These environments normally ease the development lifecycle by abstracting the low level complexities such as interacting with VR devices. However, some of these development environments are no different than using a programming language in the sense that it places the creation of the link between the logic and the simulation environment in the hands of the developer. Nevertheless, some of these development environments (Hawkes and Wray 1998, Tamberend 2003) encourage flexibility by providing a high level scripting language, thus making it feasible to be used for separating and modifying the scenario logic. One tool (Shaw et al 1992) even provides a higher level of support in the form of a simple script file for creating environments.

The commercial software environments (such as DI-Guy™ and Vega™) address the domain independence in a much better way by allowing the logic to be inserted using a graphical interface or a high-level scripting language. These succeed in achieving domain independence but usually the logic gets formatted in a proprietary format to the specific environment, thus lacking the simulation environment independence. One of the strengths of these tools is their ability to cater for a wide range of users by providing interaction methods of different levels of complexity. Such tools can be used by domain experts or scenario creators using the graphical interfaces provided. The developers also can make use of the API access provided.

KB systems (Hamman et al 2001b, Szarowicz et al 2002) are geared towards separating the logic or knowledge from the system using it. They have an inference engine to deal with retrieving the appropriate results. Furthermore, the separation also allows the modification of the knowledge independently from the simulation environment and more frequently without developer involvement, which means there is no need for recompilation of the simulation.

The drawbacks of some of the previous categories are:

- The embedding of the logic in the simulation environment makes it inflexible to change.
- The logic usually tends to be specific to the simulation environment and requires some work to be able to reuse it in a different simulation environment. This usually makes the logic created limited to a specific simulation environment.

Some of the strengths of the previous categories are:

- Interoperability between different simulation environments.
- The separation of the logic from the simulation environment shown by the KB systems.
- The decoupling accomplished using events mechanisms.
- Providing a high-level scripting language makes the interaction with the simulation environment less complex.

In comparison with the above categorization we propose a new category we call ‘simulation services’ (similar to web services). This category attempts to combine the best practices from the above categories and also avoid their drawbacks. The main goal of this category is to make the simulation ‘brain’ run as a service by linking it with the logic from one side and the simulation environment from the other side, therefore advocating independence from both sides. The independence from the simulation environment means that the ‘brain’ can be reused to service other simulation environments, possibly built using different languages, as long as they conform to common communication protocols.

OUR APPROACH

The simulation environment architecture we pursue puts an intermediary between the scenario logic and the simulation environment in an attempt to be domain and simulation environment independent simultaneously.

The distinguishing factor of the proposed intermediary lies in the way it is modularised and run as a separate service provider, used to automatically generate and service scenario behaviours to a simulation environment. The scenario behaviours are constructed from a knowledge base representing a specific domain. Participants in the simulation environment communicate their status to the intermediary and receive events to decide their course of action. The communication between the simulation and the intermediary in a second stage of development will be achieved through the use of an events service and more specifically the publish/subscribe mechanism.

KB systems have an advantage over conventional algorithmic techniques when solving complex problems. Reasoning to solve complex issues using knowledge (e.g. rule-based systems) is much simpler than reasoning about algorithms which have loops and branches (Hook 2004).

The architecture proposed is composed of three main parts: the KB, the *events space*, and the simulation environment. These need to communicate with each other in order to run the simulation environment. The role of the events mechanisms is to couple the intermediary *events space* to the simulation environment. There are many different events mechanisms such as publish/subscribe, message passing, remote procedure call (RPC), notifications, shared space and message queuing. We chose the publish/subscribe mechanism because it offers full decoupling, as explained in (Eugster et al 2003).

We suggest that using a KB system from one side and publish/subscribe event mechanism on the other side will not only achieve the proposed separation goal but it will also contribute immensely to making the intermediary module, the *events space*, fully decoupled from the simulation environment side.

EVENTS SPACE AS INTERMEDIARY

The conceptual design for the *events space* is shown in Figure 1. The directed arrows show the flow of information amongst the *events space* and the three main entities: KB, entities and relationships, and

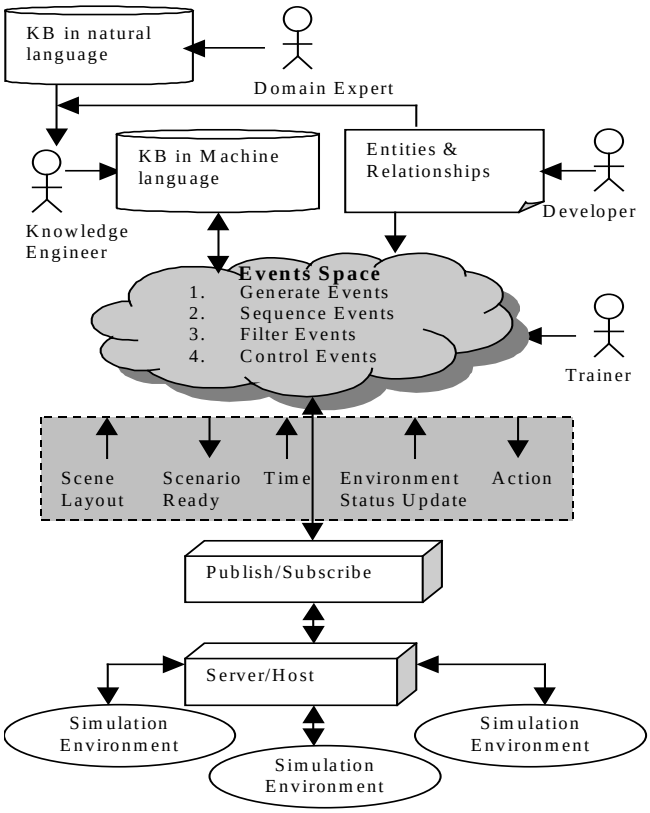


Figure 1. Events space as a link between knowledge base and simulation environments

simulation environment/s. The process of creating a virtual environment will undergo the following steps.

First, the KB is simply information elicited from domain expert/s and represented in a natural language. A Knowledge Engineer then formats such knowledge into a format that is specific to the *events space*. At this point the *events space* can access the knowledge and use it to create and control scenarios.

Second, the simulation environment developer provides the entities and relationships that represent the simulation environment engine capabilities in the form of a class diagram showing the methods, variables, etc. The format that represents the entities and relationships is again specific to the *events space*.

Third, the simulation environment sends the scene layout, time, and environment status and receives a message to state that the scenario is ready and receives events dictating what should happen next.

Once the *events space* receives the KB and entities and relationships in its format, it can then start creating various scenarios. The process of creating these scenarios starts by creating single events, which are filtered into plausible and implausible events based either on human intervention or set criteria (e.g. event duration). The filtered events are then sequenced to make up the scenario. These sequences are passed again through a filter to validate them. On the other end when the simulation environment passes its scene layout to the *events space*, a sequence of possible events from the accepted scenarios is chosen. This sequence must conform to the content of the environment layout. For example, no animation of person in pain is shown unless there is an injured person in the scene layout provided.

The scenario creation process described above can run offline and the results are stored for future use. To start the *events space* the simulation environment needs to send its simulation time and the simulation commences. The role of the *events space* switches to controlling the scenario based on the current time of the simulation and the environment status received from the various simulation environments. The *events space* generates events based on time and/or the occurrence of some behaviour (e.g. collision, user action, time increasing, etc) in one of the simulation environments. The generated events are then passed to the simulation environment.

The trainer is provided with an interface to the *events space* to allow him to filter events, create scenarios, and control scenarios created by the *events space*. The first two are done offline whereas the controlling is done at run-time where the trainer can monitor the training progress and alter the scenario to guide it towards a specific training path.

Inputs

The following sections describe the entities that interact with the *events space* in more detail and provide samples of the format.

Entities and relationships

The entities and relationships module holds a description of how events can take place in the simulation environment (i.e. the capabilities of the simulation environment - these capabilities at low level reflect the functionality of the simulation engine). The *events space* uses this information to form the events that are then passed to the simulation environment to be interpreted and acted upon. The access to the simulation environment is provided by embedding a high-level scripting language that allows run-time access to the classes' properties and methods.

KB

The knowledge base module holds the knowledge of the considered domain acquired from the domain expert by the knowledge engineer. In its natural language form it can be represented using simple rules in the form of 'IF (condition) THEN (action)'. These rules are then translated into the specific format of the *events space* by a knowledge engineer who uses the entities and relationships format guidelines, which describe the different entities, their attributes, and how they are related to each other.

Simulation Environment

The *events space* requires three types of information from the simulation environment: the scene layout, the simulation time, and changes in the environment status. The information that the *events space* sends is an acknowledgement of the readiness of the scenario and messages describing what should happen next in the simulation environment based on the current time and environment status.

The communication between the host machine that services the simulation environments and the *events space* is accomplished through the use of the publish/subscribe event mechanism. This is the only mechanism which offers full decoupling between the *events space* and the server that hosts the simulation environments. In the publish/subscribe mechanism, if two parties want to communicate then one party needs to advertise an event with the publish/subscribe service and the second party to subscribe to the advertised event.

Trainer

A trainer can interact with the *events space* in three ways: he can manually create and modify scenarios, he can filter automatically created scenarios and he can monitor currently running scenarios to

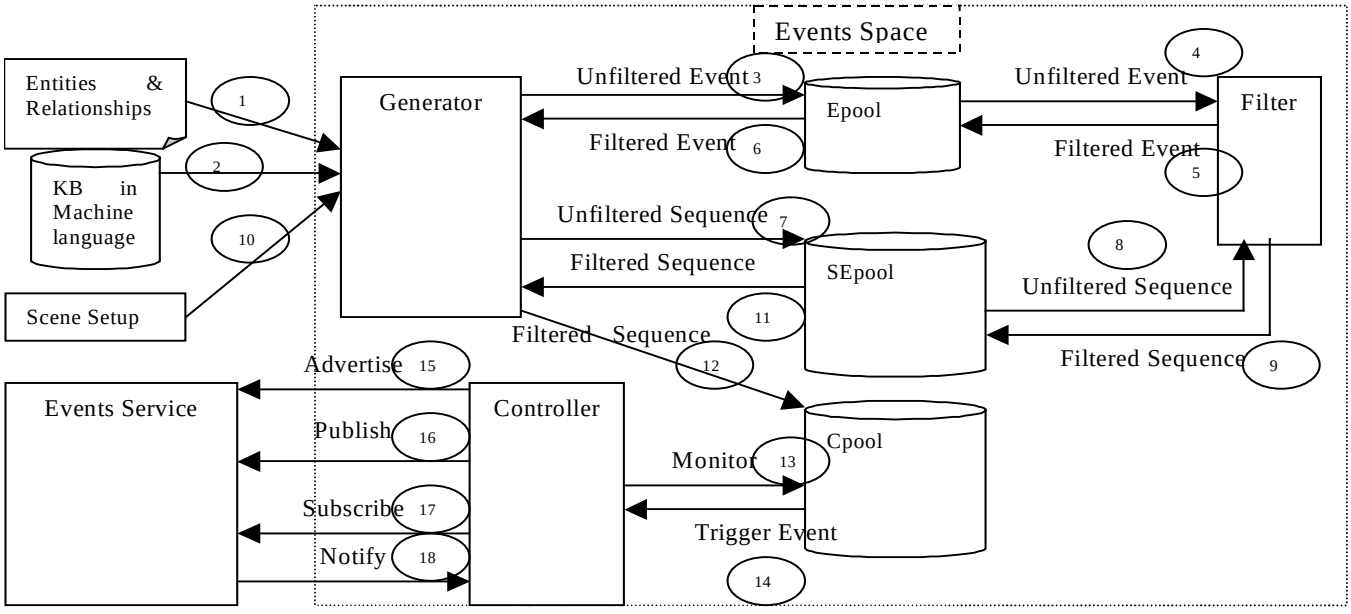


Figure 2: The internal workflow of the events space

redirect the training course to guide the training along different tracks.

Workflow

Figure 2 shows a sequenced workflow of the architecture. The dotted box shows the boundaries of the events space. The workflow can be divided into two main phases. The first phase occurs offline before the simulation environment connects to the events space. The second phase occurs after the simulation environment makes the connection.

The sequenced arrows 1 to 9 mark the first phase where the knowledge base rules are passed in machine language along with the entities and relationships to the generator component. The generator then creates events and stores them in the Epool. After that, these events are filtered by the filter and marked accepted or unaccepted. The accepted events get used by the generator to construct sequences that are placed in the SEpool. Similarly to the events, the filter again reviews these sequences and marks them acceptable or unacceptable. The next step, shown by sequenced arrows 10 to 18, occurs when the simulation sends its scene set-up to the generator to create events specific to the scene provided.

The scene set-up is used to filter sequences, and sequences that match the provided scene are then passed to the Cpool. The Cpool is used by the controller component to service events to the simulation environment.

The controller starts by receiving the synchronised event from the events service which the controller subscribed to. After synchronising the time, the controller starts monitoring the events in the Cpool to search for events consistent with the current time to be fired. The controller also checks if the preconditions of any of the events are satisfied by receiving status events from the event service. Moreover, the controller examines whether triggering an event requires other events to be triggered. To take advantage of the reasoning strength of KB systems, the controller can also be used to provide hints to the trainee by checking the Cpool, SEpool, Epool, and KB in machine language respectively for actions to the current situation.

A FIRST PROTOYPE

In the first prototype of the proposed architecture a human trainer acts as substitute for the proposed *events space.* This prototype has been used to evaluate the architecture's flexibility and suitability for collaborative virtual environments. The aim of this implementation is to simulate the behaviour of the *events space* proposed by having a domain expert doing its tasks (i.e. creating, monitoring, and controlling the simulation behaviour). Furthermore, the architecture should cater for inserting different scene layouts to make it scene independent as well. The trainer, acting as the *events space*, is able to monitor the simulation environment and trigger and control events as he sees appropriate to achieve the goals of the scenario that is running.

The two main goals to be established at this stage to address the flexibility issue are: the ability to provide a method for inserting the behaviour desired into the system without reprogramming and recompiling the application, and the ability of the prototype to allow run-time control of the behaviour inserted.

- Creating scenario behaviour: allows the trainer to create different scenarios by adding events such as adding animation paths for objects or attaching sounds to objects.
- Monitoring scenario behaviour: permits the trainer to watch the scenario unfolding by joining the scenario as an invisible participant.
- Controlling scenario behaviour: allows the trainer to stop or change the course of the simulation behaviour.

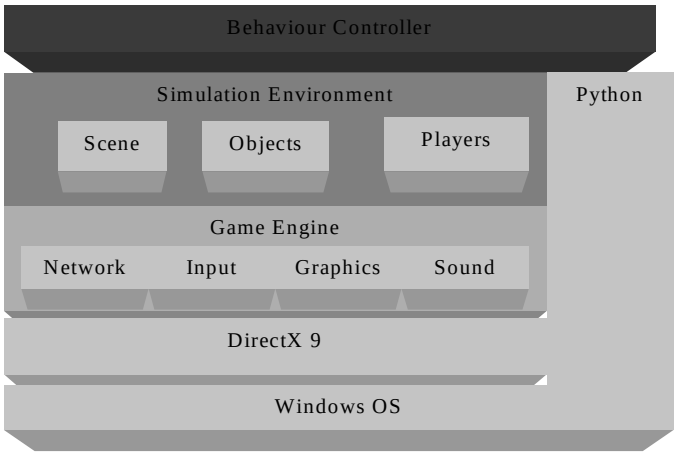
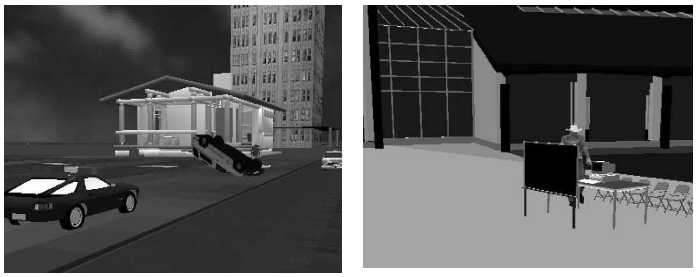


Figure 3. Simulation environment architecture



(a) (b)

Figure 4. Vehicle accident investigation (a), virtual lecturing (b)

- Inserting a scene layout: permits the trainer to insert different scenes making the architecture scene independent.

To enable the manipulation of the scenario behaviour, a high-level scripting language has been embedded in the architecture as shown in Figure 3. The language used in our prototype is Python, which allows for easy replication of the game engine classes that are based on an object-oriented approach using C++. Furthermore, the other advantage scripting provides to this prototype is the ability to dynamically load code (Hook 2004) which can be used to insert and control the behaviour. Figure 3 shows how the scripting language is embedded across three architectural layers (DirectX 9.0, simulation engine, and simulation environment), which allows it to access any of them. A game engine has been built on top of DirectX to abstract all of its complexities. The simulation environment lies above the game engine and holds the objects, the players, and the scene settings. The top layer is the behaviour controller which allows run-time access to the simulation environment and allows the trainer to broadcast events inserting or changing the behaviour of any objects, players, or part of the simulation environment.

The scenario creation passes through the following steps: model creation, scene set-up, and scenario configuration. The models are created using a 3D drawing tool; we have used 3D Studio Max 6.0. Optimised models are then exported to the Microsoft DirectX file format, which is then used by the Scenario Creator. Subsequently, the scene layout is created by positioning and orienting the objects in the scene.

The Scenario Creator inserts the objects' information into a database contains five tables, three of which are specific to the scenario (games, players, and scene objects) and the other two hold reusable general data (meshes and groups). The scene tab allows the insertion and positioning of objects in the environment. Titles are assigned to each object to ease their identification. Each object created has also its own unique identification number which is stored in the SceneObjects table.

Two scenarios were deployed on the prototype architecture to show that the scenario logic is no longer embedded in the simulation environment and that it can be inserted and controlled by an outside component (the trainer using an interface). The first scenario aims at training new police officers on how to investigate and deal with the aftermath of a traffic accident. The second scenario is a virtual classroom which allows participants to join a single environment and communicate using voice. Figure 4 shows the two scenarios. Figure 5 shows one of the scenarios running on a Reflex set-up at the University of Sheffield.

The vehicle accident scenario was run on six subjects from Dubai Police who all had the same training background which consists of four years of police academic college and two specialized training courses in vehicle accidents investigation. The six subjects where chosen by the trainer. During the run of the experiment we noticed that all the trainees made good use of the navigation methods and managed to investigate the accident scene. They also made good use of the headset facility in identifying the drivers and communicating with the operation room operator. One of the trainers said that practical training has shown him clearly the trainees' weaknesses which were not obvious to him while conducting the theoretical training.

CONCLUSIONS

The separation of the domain knowledge and the ability to control it at run-time were the two primary objectives of the first prototype. The separation has been achieved by the use of a high-level language (Python) to create a layer on top of the simulation environment to act as an interface. This means that behaviours can be inserted and controlled at run-time. The abstraction achieved by the scripting language made controlling the application behaviour much easier and dynamically loadable at run-time. Moreover, using an existing language rather than building our own reduced the development time and eliminated the need to build a parser and evaluate it. The other advantage of this approach is that it allows easier commands to be added making the use of the simulation engine simpler.

Creating and running two different scenarios on the prototype simulation environment has demonstrated the flexibility of the approach since it managed to detach the domain knowledge from the simulation environment and allow its control at run-time. These results are promising. Proving full flexibility would require more tests to be carried out involving scenarios from different domains. The observation that can be made about the flexibility of the prototype is that the more the simulation environment functionality is exposed through the scripting language the more flexibility is achieved. This means if the full simulation environment functionality is exposed, then the system can be labeled fully flexible according to our description of flexibility, i.e. the ability to insert and control behaviours at run-time.

We have shown that our prototype architecture can handle the running of multiple environments while allowing a trainer to monitor the simulation and trigger events to create the desired scenario. The next stage will replace the trainer by the architecture proposed in Figure 1.

REFERENCES

Akerberg, O., Svensson, H., Schulz, B., Nugues, P. *CarSim: An Automatic 3D Text-to-Scene Conversion System Applied to Road Accident Reports*. Research Notes and Demonstrations Conference Companion, 10th Conference of the European Chapter of the Association of Computational Linguistics, 2003. http://citeseer.nj.nec.com/563862.html

Cruz-Neira, C., Bierbaum, A., Hartling, P., Meinert, K., Just, C. *VR Juggler – An Open Source Platform for Virtual Reality Applications*. AIAA 2002 Aerospace Science Conference, Reno, NV, January 2002.

Dachselt, R. *CONTIGRA Towards a Document-based Approach to 3D Components*. Workshop 'Structured Design of Virtual Environments and 3D-Components' at the ACM Web3D 2001 Symposium.

Davis, W., Moeller, G. *The High Level Architecture: is there a better way*. In Proceeding of the 1999 Winter Simulation Conference

Figure 5. Accident scenario running on Reflex Studio set-up.

Drew, R., Morris, D., Dew, P., Leigh, C.*A System Architecture For Supporting Event Based Interaction And Information Access*.http://citeseer.nj.nec.com/370116.html

Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A. *The many faces of publish/subscribe.* In ACM Computing Surveys (CSUR), volume 35, issue 2 (June 2003), pages: 114 - 131

Hamman, M., Wilkins, D. C., Carbonari, R., Mueller, C. *DC-Train 4.0 Instructor's Manual*. Knowledge Systems Lab Report UIUC-BI-KBS-2001-0040. Beckman Institute, University of Illinois, Urbana-Champaign. November, 2001.

Hamman, M., LeMentec, J. C., Wilkins, D. C., *Design Requirements for DC-Train 4.0*. Knowledge Systems Lab Report UIUC-BI-KBS-2001-0029. Beckman Institute, University of Illinois, Urbana-Champaign. February 2001.

Hawkes, R., Wray, M. *"LivingSpace: A Living Worlds Implementation using an Event-based Architecture"*. HPL-98-181, Extended Enterprise Laboratory, 1998.

Hook, B. *The Secret Life Of Game Scripting*. Feb, 2004 http://bookofhook.com/Article/GameDevelopment/TheSecretLifeofGameScript.html

Romano, D.M. *Features that Enhance the Learning of Collaborative Decision Making Skills under Stress in Virtual Dynamic Environments*. Ph.D.thesis, Computer Based Learning, University of Leeds, UK, August 2001.

Shaw, C. Liang, J, Green, M. Sun, Y. *The Decoupled Simulation Model for Virtual Reality Systems. In* Human Factors in Computing Systems CHI'92 Conference Proceedings, pages 321-328, Monterey, California, May 1992. ACM SIGCHI.

Smith, R. *Essential techniques for military modeling and simulation*. Proceedings of the 30th conference on winter simulation, 1998, pages: 805 - 812 ISBN:0-7803-5134-7

Szarowicz, A., Forte, P., Amiguet-Vercher, J., Gelepithis, P. *Application of Autonomous Agents for Crowd Scene Generation*. 2nd Hellenic Conference on AI SETN-02, vol. 2 April 11-12, Thessaloniki, Greece, 2002

Tamberend, H. *Avocado: A Distributed Virtual Environment Framework*. Ph.D.thesis, University of Bielefeld, 2003.

Wang, Q. Green, M, Shaw, C. *EM – An Environment Manager for Building Networked Virtual Environments*. IEEE Virtual Reality Annual International Symposium (VRAIS 95), *pages 11-18, Research Triangle Park, North Carolina, March 11-15, 1995, IEEE.*

Williams, R, J. *A Simulation Environment to Support Training for Large Scale Command and Control Tasks*. Ph.D. thesis, School of Computer Studies, University of Leeds, UK, December 1995.

Wright, I. P., Marshall, J. A. R. *RC++: a rule-based language for game AI*. In: Proceedings of the First International Conference on Intelligent Games and Simulation (GAME-ON 2000). SCS Europe BVBA