The Challenges of Designing an Open Extensible Software Toolkit for Distributed Simulation

Randall E. Barker
James R. Damiano
Gregg D. Marcus
Reality by Design Inc.
354 West Cummings Park
WOBURN, MA  01801-6335

Paul T. Barham
Reality by Design Inc.
910 Major Sherman Lane, Suite 205
MONTEREY, CA  93940-1054

Keywords: Simulation, HLA, C++, Object-Oriented, DLL, Distributed Simulation, Open Standards, Simulation Architecture

ABSTRACT: Reality by Design (RBD) has developed an extensible software component toolkit for building distributed simulation applications. This object-oriented toolkit is designed to utilize open standards, dynamically loadable libraries, and well-defined interfaces to provide users with a tool to implement high performance, maintainable, cross-platform distributed simulations. This paper presents the design and discusses the unique challenges faced during the implementation of this distributed simulation toolkit.

1. Introduction

Over the last decade, distributed simulation has progressed from narrowly defined systems of tightly coupled simulators to a new generation of more diverse simulations interconnected more flexibly at the interface level. Early generations of distributed simulation were based on strict adherence to network protocols with predefined data content requirements. The SIMulation NETwork (SIMNET) [1] and, later, Distributed Interactive Simulation (DIS) [2] were significant examples of this approach to distributed simulation. The new generation, as embodied in the High Level Architecture (HLA) [3], represents a major paradigm shift. Simulations are now defined at their interface level rather than by strict data requirements. This new approach encourages the development of a vast array of new types of interoperable simulations. However, with this freedom comes a new level of complexity and a clear need for a software developer’s kit to bring a degree of order to the complexity at the new frontier of distributed simulation.

Reality by Design (RBD) has developed such a toolkit to facilitate in the development and maintenance of distributed simulation applications [4,5]. Designing a distributed simulation toolkit that addresses the needs of the HLA community, legacy DIS simulations, and future innovations presents a set of challenges that can be divided into several categories. In this paper we discuss challenges that were encountered in the requirements, programming paradigm selection, architecture, and implementation stages of this development process. Application examples are included to clarify the development process. The overarching goal was to design a truly powerful, versatile and usable distributed simulation toolkit.

2. Functional Requirements

The initial challenge in the development of our toolkit was to define a set of functional requirements which would enable developers to utilize available technology to satisfy the broadest range of end user requirements.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of Use</td>
<td>To promote efficiency and productivity in the development process.</td>
</tr>
<tr>
<td>High Performance</td>
<td>To support real-time, large-scale simulations.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>To minimize life-cycle cost and facilitate the incorporation of new domain knowledge into the simulation.</td>
</tr>
<tr>
<td>Scalability</td>
<td>To support new requirements and ever increasing scope.</td>
</tr>
<tr>
<td>Cross-Platform Support</td>
<td>To support multiple operating systems to enable maximum selection of available hardware.</td>
</tr>
<tr>
<td>Open Standards Support</td>
<td>To gain maximum leverage from existing and future simulation R&amp;D and products.</td>
</tr>
<tr>
<td>Inclusion of Key Distributed Simulation Functionality</td>
<td>To provide and encapsulate common simulation functionality in order to avoid redundant development efforts, reduce errors, and optimize computer memory usage.</td>
</tr>
</tbody>
</table>

Table 1. Functional Requirements
Simulation hardware, software, and end user requirements are constantly changing. The toolkit must be able to pull together available technology and integrate it into a seamless package that solves a particular distributed simulation problem. Refining this general requirement into the specific requirements listed in Table 1 provided the framework for subsequent development of the toolkit.

3. Programming Paradigm

Choosing the proper programming paradigm determines the ease with which the toolkit is able to meet its requirements. The closer the paradigm matches the problem-space, the easier it will be to solve the problem. For example, a command line interface to a graphics program will require far greater effort from the user than a graphical user interface to produce the same results. The challenge here is to choose a programming paradigm that allows the toolkit to easily fulfill the requirements defined in Table 1.

The decision was made to base the toolkit on a dynamically reconfigurable and extendable object-oriented paradigm. The fundamental principles of this paradigm are explained in Table 2.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardize at the Interface Level</td>
<td>Allows independent development of basic modules and systems composed of these modules. Supports ease of use, maintainability, scalability, and cross-platform operation.</td>
</tr>
<tr>
<td>Support Modularity</td>
<td>Enables functional disassembly of the simulation problem space to encapsulate key simulation functionality. Supports ease of use, maintainability, scalability, and cross-platform operation.</td>
</tr>
<tr>
<td>Minimize Dependencies Between Modules</td>
<td>Simplifies code changes enhancing ease of use, maintainability, scalability, and cross-platform operation.</td>
</tr>
<tr>
<td>Support Dynamic Reconfiguration</td>
<td>Allows systems to be reconfigured at runtime to improve usability, maintainability, and scalability.</td>
</tr>
<tr>
<td>Support Extension of Functionality</td>
<td>Provides a mechanism to extend functionality, which enhances maintainability and scalability.</td>
</tr>
</tbody>
</table>

Table 2. Principles of the Dynamically-Reconfigurable and Extendable Object-Oriented Paradigm

Looking back at the functional requirements, it is apparent that this programming paradigm does not specifically address high performance or open standards. These requirements are highly dependent on the chosen architecture and implementation and will be covered in the following sections.

4. Architecture

Designing the toolkit architecture was the primary challenge. Functional boundaries and methodologies, defined in the requirements and programming paradigm, were crafted into a blueprint that would naturally support the implementation. The process of developing the architecture and implementing it was iterative. Lessons learned in the implementation were continuously folded back into the architecture. Throughout this process, the programming paradigm guided the development. Standardizing at the interface level, modular code and dynamic reconfiguration were guiding principles that resulted in the architecture depicted in Figure 1.

Figure 1. Conceptual Drawing of the Toolkit Architecture

The architecture is based on four key component types, which include the Operating System Abstraction Layer (OSAL), Foundation Classes, Modules, and Plug-Ins. The toolkit dynamically loads all of these components at runtime. This is possible because most link symbols are resolved at runtime instead of while linking. This dynamic loading makes it possible to drastically change a simulation’s configuration at startup. The following sections describe each of the key component types.
4.1 Operating System Abstraction Layer (OSAL)

The operating system abstraction layer intermediates between the operating system and the remainder of the code to facilitate cross-platform compilation, high-performance code execution, scalability, ease of use, and maintenance. It is a set of software utilities that provides a common set of operating system (OS) level services while encapsulating the underlying OS implementation. The types of services provided include timing services, synchronization, and the dynamic loading of runtime libraries. Thread and multi-process handling enhance both performance and scalability. By exclusively utilizing the OS level services of the OSAL, higher level libraries remain OS- and platform-independent, which results in a toolkit that is easier to use and maintain.

4.2 Foundation Classes

The Foundation Classes are the unifying software infrastructure that provides the common set of services needed by all other components of the architecture. These services include data communications between the components, conflict resolution, scheduling, and data capture. Written on top of the OSAL, the Foundation Classes are platform-independent standardized classes that make the toolkit easier to use and simulations developed with the toolkit easier to maintain.

4.3 Modules

Modules are the primary mechanism that the architecture uses to provide ease of use and maintainability and to contain key simulation functionality. The programming paradigms of standardizing at the interface level, modular code, and minimal dependencies guided the development of the modules. A module is a reusable, modular software library that provides specific functionality to the higher-level simulation system through a well-defined and published interface.

A module uses a specific implementation to provide the services defined by the standardized interface. The guiding principle is to design modules that contain essential simulation functionality in a package that depends on a minimal amount of information from the remainder of the simulation. Modules of the same type are interchangeable because they are written to satisfy the same standardized interface. Different standardized interfaces have been developed to address logical segments of distributed simulation functionality. These logical segments are based on an evaluation of the taxonomy of a wide range of virtual environment and DoD simulation systems from the live, virtual, and constructive domains. Table 3 summarizes these standardized module interfaces.

Standardizing at the interface level allows simulation developers to substitute their own module implementation for those that are included in the toolkit. This allows for simulations to scale up addressing unforeseen requirements.

A second layer interface, appended to the back end of the modules provided with the toolkit, allows developers to extend the functionality of the included module set. This extension interface provides for additional features that may be needed for a specific simulation requirement. The extension must use an implementation that is consistent and compatible with the module it is extending.

### Table 3. Standardized Module Interface Definitions

<table>
<thead>
<tr>
<th>Interface</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioFX</td>
<td>Responsible for providing spatialized audio information.</td>
</tr>
<tr>
<td>Entity</td>
<td>Responsible for gathering, maintaining and transmitting entity (object) information including: entity attribute ownership and content and remote entity approximation (e.g., dead reckoning).</td>
</tr>
<tr>
<td>Environment</td>
<td>Responsible for managing the state of the virtual world and providing state information.</td>
</tr>
<tr>
<td>Event</td>
<td>Responsible for gathering, maintaining and transmitting fire, detonation and collision information.</td>
</tr>
<tr>
<td>Human Computer Interface (HCI)</td>
<td>Provides access to input and output (I/O) devices.</td>
</tr>
<tr>
<td>Network</td>
<td>Responsible for capturing and transmitting all simulation network information.</td>
</tr>
<tr>
<td>Render</td>
<td>Responsible for providing 3D visual information and 2D overlays.</td>
</tr>
<tr>
<td>3D Sensor</td>
<td>Provides access to 3D tracking devices.</td>
</tr>
</tbody>
</table>

4.4 Plug-Ins

A plug-in is a reusable, modular software library which provides specific functionality, operates as a subsystem of a specific application, and is implemented on top of the foundation classes, modules, and OSAL. Plug-ins differ from modules in that plug-ins are implemented using the functionality provided by the foundation classes, modules, and OSAL and are thus platform and OS independent but toolkit architecture dependent. In contrast, modules are implemented at a much lower level on top of the OSAL and OS of the target platform and are thus somewhat platform and OS specific but are reusable outside of the toolkit architecture.

Plug-ins promote ease of use and facilitate maintenance by providing the simulation developer with a standardized interface. All plug-ins have the same interface that consists of simple member functions that allow the developer to discover, initialize, synchronize (advance one simulation cycle), and shut down modules. This plug-in interface is the developer's window into the dynamic runtime configurable functionality of the toolkit. A collection of plug-ins that interact to solve a distributed simulation problem is called an application.
5. Implementation

Although the requirements, programming paradigm, and architecture have laid the groundwork for a truly functional distributed simulation toolkit, it is the implementation that determines performance. The challenge at this stage is to choose the appropriate open standards, target platforms, and coding techniques so that the resulting toolkit is easy to use, high performance, maintainable, and scalable. The biggest challenge is to keep inter-module dependencies to a minimum. The following sections highlight the major implementation challenges and solutions from choice of a programming language through the implementation of each module.

5.1 Programming Language

The most fundamental implementation decision is the choice of a programming language. The programming paradigm chosen requires a language that supports object-oriented programming techniques. C++ was chosen because it is an international standard, has a consistent implementation across platforms, is extremely efficient, and supports object-oriented programming.

5.2 ASCII Text Configuration Files

Virtually every option can be set in a text file with no requirement to edit or recompile source code. This drastically enhances ease of use and maintenance of the code.

5.3 OSAL Implementation

The OSAL is implemented as a thin layer of object-oriented components common to most modern-day operating systems. OSAL implementations determine which platforms will be supported by the toolkit. The toolkit includes two implementations of the OSAL. A Win32 implementation is included to support operation with Windows 95/98/NT/2000 platforms. A POSIX implementation supports Linux, IRIX™ and Solaris™ platforms. Other platforms can be supported with new implementations of the OSAL. The OSAL is characterized by its data types, interface and associated classes.

Data types define the simple types utilized inside of the OSAL namespace. All of the simple datatypes utilized in the toolkit are defined inside of the OSAL namespace to help avoid namespace collisions. The OSAL interface is characterized by a set of member functions. Some of these member functions are factory functions. A factory function abstracts out the true type of the object instantiated inside of the OSAL and returns a handle to the base class. This eliminates the need for the developer to tailor function calls for the underlying platform or architecture. For example if you are running under Windows and you create an osal::semaphore, the OSAL actually creates an osal::semaphoreWin32, which gets returned as its base type, which is osal::semaphore. Therefore, the developer does not need to be concerned with the specific details of accessing the system-level services of any one platform.

Each associated class encapsulates a particular piece of operating system functionality. Associated classes are classes that can only be instantiated by invoking the factory functions within the OSAL. For example osal::thread provides an abstraction layer between the application and the operating system implementation of the threading process.

The decision to support multi-threading and semaphores greatly increases performance and scalability. These two capabilities allow a simulation to run on multiple processors to parallelize the processing of information.

5.4 Foundation Classes Implementation

Foundation classes are an extensive collection of C++ classes that pull all of the other toolkit components together, store data, run applications, and provide basic functionality. The foundation classes are partitioned into a set of component libraries that include:
- dseCoordConverter: A single library designed to perform standard coordinate system conversions
- dseGL: A general-purpose 3D graphics library
- dseFoundation: a collection of libraries that provide basic simulation functionality. Table 4 describes the functionality of each logical grouping of dseFoundation components. Each logical grouping contains numerous member functions that are not individually listed here.

<table>
<thead>
<tr>
<th>Logical Grouping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dseAtomic</td>
<td>Describes an entity at its most basic level (e.g., orientation, position, state).</td>
</tr>
<tr>
<td>dseConfig</td>
<td>Provides classes that make up the configuration database for initialization of all components.</td>
</tr>
<tr>
<td>dseConfigUtil</td>
<td>Allows for the formatted printing of the contents of a namespace to an ostream.</td>
</tr>
<tr>
<td>dseData Marshal</td>
<td>Performs byte swapping on buffers. Both unpacks and packs them.</td>
</tr>
<tr>
<td>dseExcept</td>
<td>Defines all of the different exception classes.</td>
</tr>
<tr>
<td>dseIsectUtil</td>
<td>Provides objects used for intersection testing.</td>
</tr>
<tr>
<td>dseLibrary</td>
<td>Manages dynamically loadable libraries.</td>
</tr>
<tr>
<td>dseMath</td>
<td>Provides math classes that are commonly used in simulation and not included in C++ (e.g., vectors, matrices).</td>
</tr>
<tr>
<td>dseReporter</td>
<td>Provides error and information reporting mechanisms.</td>
</tr>
<tr>
<td>dseRuntime</td>
<td>Contains several variables that need to be configured such as state, object type and ID, etc.</td>
</tr>
<tr>
<td>dseScheduler</td>
<td>Manages time scheduling.</td>
</tr>
<tr>
<td>dseTemplate</td>
<td>Provides templates for hash table of strings and unsigned integers.</td>
</tr>
</tbody>
</table>

Table 4. dseFoundation Classes
Foundation classes sit on top of the OSAL and can be utilized in virtually all of the other toolkit components. Standardizing all of these components helps to make the toolkit easier to use and maintain. Careful implementation of these classes contributes to the performance and scalability of any simulation that incorporates them.

5.5 Module Implementations

Module interfaces define the simulation functionality that is to be embedded in each module. The actual implementation of each module is largely responsible for the minimization of interdependencies and incorporation of open standards into the toolkit. Module extension interfaces add to the scalability and maintainability of simulations. The following sections describe the implementation of each module (see Figure 1).

5.5.1 AudioFX Module

The AudioFX module has a glue layer that contains an interface to the higher levels of a simulation. Additionally, one extension interface is defined to support different extension implementations. Extension implementations can be designed to support: different fidelities (diotic, stereo, binaural 3D), different processing models (DirectSound™, Focal Point™, Aureal™, Creative™) and different presentation devices (headphones, dual speaker, quad speaker, 5 speaker). The modularity of the AudioFX module, combined with the well-defined and published interfaces, allows selection of the module implementation that yields the highest fidelity and performance for a given hardware configuration.

5.5.2 Entity Module

The Entity module communicates with the rest of the simulation through a distributed object database interface that allows the higher levels of the simulation to query information about entities and their attributes, and to define and set attributes for locally-controlled entities. Custom dead reckoning algorithms can be enabled with a dead reckoning extension. This simple design is responsible for gathering, maintaining and transmitting entity (object) information.

5.5.3 Environment Module

The Environment module is responsible for managing the state of the virtual world and providing state information to the higher levels of the simulation through its well-defined interface. State information is derived from intersection testing using line segments and rays. The initial version of the environment module maintains a static state of the environment based on the initial conditions of the simulation. The design of the environment module assumes that future versions will be required to handle changes to the environment based on locally simulated events as well as networked events. Therefore, the interface developed for the synthetic environment module provides set and query operations for an evolving underlying environment. The synthetic environment includes data concerning the terrain (e.g., surface material, trafficability, etc.), atmosphere (e.g., weather, visibility, etc.), and feature models (e.g., buildings, trees, power lines, etc.). Extensions that support crating, multi-state buildings, and any other operation that requires the modification of a polygonal surface based on a stimulus will be created as needed.

The Environment module accesses synthetic environment data through the dseGL foundation class. dseGL provides a standard mechanism for reuse and sharing of synthetic environment databases (including, for example, terrain, cultural models, and weather), regardless of data format or size. By accessing synthetic environment data through the dseGL interface, the toolkit supports the import and dynamic paging of synthetic environment data. dseGL supports the Synthetic Environment Data Representation & Interchange Specification (SEDRIS) open standard. SEDRIS allows for the import of any standard DoD format data set [6].

5.5.4 Event Module

The Event module is responsible for gathering, maintaining and transmitting fire, detonation and collision event information. Event parameters are either received all at once from a plug-in or piecemeal from the network. Event parameters are always transmitted to the network one parameter at a time.

5.5.5 Human Computer Interface (HCI) Module

The Human Computer Interface (HCI) module provides a standardized interface for the higher levels of a simulation to access input and output (I/O) devices. Extensions are provided to interface with various controllers (joystick, SpaceOrb™, etc.). The module is designed to dynamically load the appropriate controller extension at startup so that changing controllers does not require recompiling or linking code.

5.5.6 Network Module

The Network module is responsible for capturing and transmitting all simulation network information and providing this information to the higher levels via a well-defined interface. The interface simply registers and unregisters adapters that are passed in from the higher level simulation. Adapters handle network data parsing and the packaging of simulation data for transmission over the network. A large set of adapters is supplied to enable operation with a broad range of network protocols including DIS and HLA Run Time Infrastructure (RTI) open standards. Network protocol changes and Federation Object Model (FOM) agility are accommodated by simply writing and registering new adapters. Two network modules ship with the toolkit. The HLA module uses an HLA adapter set and writes to the RTI interface. The User Datagram Protocol (UDP) module supports DIS and custom protocols with appropriate adapter sets. UDP packets are broadcast, multicast or unicast onto the network[7].

5.5.7 Render Module

The Render module is responsible for providing 3D visual information and 2D overlays to the application
user. A glue layer implements a chosen graphics library such as dseGL, which in turn supports OpenGL™ or Direct3D™. Other implementations can be written to support Performer™ or any other custom graphics library. The current implementation has one extension. The DiGuy™ extension supports Boston Dynamics, Inc. (BDI) Di-Guy™, dismounted infantry animation package. Synthetic environment extensions will support smoke, fire and cloud rendering among other effects as the need arises. The capabilities of each render module implementation are specialized and limited to the underlying hardware functionality. Each module has a standardized interface for the higher levels of a simulation to produce output to the user. The modularity of the Render module, combined with the well-defined and published interface, allows selection of the module implementation that yields the highest fidelity and performance for a given hardware configuration.

5.5.8 3D Sensor Module

The 3D Sensor module provides a standardized interface for the higher levels of a simulation to access 3D tracking devices. Extensions are provided for various trackers (InterSense™, Ascension™, etc.). The module is designed to dynamically load the appropriate tracker extension at startup so that changing trackers does not require recompiling or linking code.

5.6 Plug-In Implementation

Several plug-ins have been implemented to tie the toolkit together. These include the Default plug-in which is used to set up inter-module and plug-in communications, the Render plug-in which links other modules to the render module, the AudioFX plug-in which links other modules to the AudioFX module, and the Object Time-Out plug-in which is used to remove DIS entities that have not been updated for greater than five seconds. The user is responsible for implementing a collection of plug-ins to create an application.

6. Sample Application

Applications are composed of a collection of plug-ins. Each plug-in encapsulates a logical portion of the application’s functionality. Plug-ins have the ability to communicate with each other; therefore, it is up to the application designer to manage the dependencies that are set up between plug-ins. Module interdependencies are minimal due to the design of the toolkit. The following example demonstrates how a modular application is constructed with the toolkit.

A common application in the distributed simulation domain is the stealth. A stealth application allows the user to see the virtual world without being detected by other simulations on the network. A basic stealth allows the user to fly around the virtual world using a joystick. The viewpoint is completely controlled by the joystick. It is possible to fly through buildings and underground. A distributed simulation generates virtual world activity. Figure 2 shows the basic configuration of a stealth application.

The stealth application initializes itself at startup, loops to process information and is then shut down. The default plug-in sets up all of the connections between plug-ins and modules at startup and loads network adapters into the Network module which then talks directly to the Entity and Event modules. Simulation data enters the Network module and is routed, via a set of adapters, to the Entity and Event modules. The Entity module keeps track of object positions in the virtual world and passes this information to the Render module. The Event module keeps track of events such as detonations and collisions and passes this information to the Render module. The HCI module collects data from the joystick and passes this information to the Stealth plug-in. The Stealth plug-in determines the eyepoint based on joystick input and passes this to the Render module. The Render module combines all of this information with the terrain database and renders an image that is output to a monitor or other suitable display device. This basic stealth can be upgraded in a modular fashion using plug-ins developed with the toolkit as well as additional modules.
plug-in, which in turn queries the Environment module for an intersection test. If the eyepoint does not intersect the terrain then it is passed back to the Stealth plug-in unchanged. However, if the requested eyepoint intersects the terrain, the Stealth Movement plug-in passes back an eyepoint that is adjusted so that it maintains a chosen offset from the terrain. The Stealth plug-in then passes the adjusted or unadjusted eyepoint to the Render module, which renders the scene with input from the terrain database as well as from the Entity and Event modules, as depicted in Figure 3.

![Figure 3. Upgraded Stealth Architecture](image-url)

The Stealth can be operated with or without movement modes. The upgraded Stealth can simply not load the Stealth Movement plug-in and Environment module and the application is back to the basic stealth functionality with no modifications to the basic code. Other modular components to implement 2D overlays, remote eyepoint control via another simulation on the network, and audio effects can all be added and will function individually or in any grouping.

The stealth application will load the required components at runtime in order to operate with the desired capabilities. This modular upgrade path makes it easy to scale an application up or down. Functionality can be added or sometimes, more importantly, removed with little effort after the plug-ins have been developed. This is a valuable application maintenance feature.

7. Summary

The challenges of designing a distributed simulation toolkit have been presented in a format that walks the reader through the design process. An example application was developed and enhanced using the toolkit in order to demonstrate the design and maintenance process. The overarching goal was to design a truly powerful, versatile and usable distributed simulation toolkit.

The toolkit was designed with a logical progression from requirements to implementation. Functional requirements were defined to frame the development of a toolkit that would enable simulation developers to satisfy a broad range of end user requirements. Next, a dynamically reconfigurable and extendable object-oriented programming paradigm was developed to ensure that the toolkit architecture would match the problem space defined in the requirements. The programming paradigm was used to derive an architecture composed of four key components with each component being defined at its interface level. The four key components of the implemented toolkit architecture are an Operating System Abstraction Layer (OSAL), Foundation Classes, Modules, and Plug-ins.

As a first exercise, the toolkit was used to develop a basic stealth application. The basic stealth utilizes a core set of modules and user-developed plug-ins to implement an application that is capable of visualizing a distributed simulation exercise. The stealth gives the user a three-dimensional view of the exercise and allows the user to move around the virtual world with a joystick. Motion in the virtual world is unrestricted, allowing users to move under the ground.

As a second exercise, the basic stealth was upgraded using the architecture’s modular approach. The upgraded version of the stealth enhances the basic stealth with an additional plug-in and module that allow the viewpoint to hug the ground when moving. This upgrade can be selected at runtime. If it is not selected, the additional components are not loaded. This feature allows simulations built with the toolkit to be configured at runtime in order to maximize both performance and memory usage.

The commercial version of the toolkit is called SimStorm SDK™ and is available from Reality by Design (RBD). RBD has and continues to use SimStorm SDK to create commercially available advanced man-in-the-loop virtual simulators, weapons inspection training tools, the RBD Stealth Pro™, and prototype games.

8. References


9. Author Biographies

RANDALL E. BARKER is a simulation engineer with Reality by Design. He is a key contributor on the design, architecture, and implementation of software simulation products. Mr. Barker is the lead developer on the DSE project. He is the primary engineer responsible for converting the RBD HMMWV DIS driving simulator to be the first certified HLA Federate. Mr. Barker worked in many areas of visual simulation on the Naval Postgraduate School’s NPSNET-IV program. He holds a Bachelor of Science in computer science from the University of California, Davis.

PAUL T. BARHAM is a Lead Simulation Architect and Principal Investigator at Reality by Design. Mr. Barham is responsible for new projects and initiatives at RBD. A large focus of his efforts relates to individual combatant simulation and the High Level Architecture. Prior to his work at RBD, Mr. Barham was the senior architect and lead programmer on the NPSNET-IV Project at the Naval Postgraduate School (NPS). Before coming to NPS, he taught Software Design and 3D Computer Graphics courses at N. C. State University, where he earned his Masters degree in Computer Science. He is a member of ACM, ACM SIGGRAPH, IEEE, IEEE Computer Society and SID.

JAMES R. DAMIANO is a virtual environments technology specialist and content developer at Reality by Design. His background includes visual simulation project management and systems integration as well as computer modeling and graphic design for virtual environments. He has developed or co-developed systems featuring human anatomy, planetary exploration, archeological site reconstruction, and telepresence. He has exhibited virtual environments systems on Capitol Hill in Washington, DC and at leading science and technology museums in the U.S. and Canada. Mr. Damiano has a Bachelor of Arts in Economics from Cornell University.

GREGG D. MARCUS is a staff systems engineer and product manager at Reality by Design, Inc. He is responsible for product requirements, systems engineering and commercialization. Mr. Marcus’ background includes over a decade of performing requirements analysis, architecture, design, test and marketing for a variety of simulation systems. He has been instrumental in the productization of the SimStorm SDK and SEDRIS research. Prior to working for RBD, Mr. Marcus was a senior simulation systems engineer at Lockheed/Martin Advanced Simulation Center. He performed systems engineering and business development for numerous HLA and DIS projects for corporate marketing, communications, product design, training and C4I applications. Mr. Marcus has a Master of Science in Computer and Electrical Engineering from the University of Massachusetts and a Bachelor of Science in Electrical Engineering from Worcester Polytechnic Institute.