SOFTWARE DESIGN DOCUMENT (SDD)

Delta3D - After Action Review (AAR)

Volume 1 - Logging and After Action Review

Version 1.6

April 17, 2006

Authored by:
BMH Operation of
Alion Science and Technologies
5365 Robin Hood Road, Suite 100
Norfolk, VA 23513-2416
Telephone: (757) 857-5670
Fax: (757) 857-6781

For more information on Delta3D, go to the website at www.delta3D.org or contact Curtiss Murphy at cmmurphy@alionscience.com, (757)-857-5670, ext 308.

Copyright © 2006 by Alion Science and Technology

Table of Contents

1 Overview **1**

1.1 Logging *1*

2 Component Design **2**

2.1 Server Logger Component *3*

2.1.1 Class Diagram 3

2.1.2 Class Descriptions 3

2.2 LogController Component *5*

2.3 BinaryLogStream *6*

3 Messages **7**

3.1 Logger Messages *7*

4 Log File Format **8**

4.1 DLM File Structure (Delta Logger Messages) *9*

4.1.1 Header 9

4.1.2 Game Message 10

4.2 DLI File Structure (Delta Logger Index) *10*

4.2.1 Header 10

4.2.2 Tags and Keyframes 11

5 Demonstration Application(s) and Unit Tests **12**

6 Notes **12**

Table of Figures

Figure 1 Logger Components Overview **2**
Figure 2 ServerLoggerComponent Class Diagram **3**
Figure 3 LogController Class Diagram **5**
Figure 4 BinaryLogStream Class Diagram **6**

List of Tables

Table 1 Logger Messages **8**
Table 2 Log File Format – Basic Data Types **9**
Table 3 Messages Database – Header **10**
Table 4 Messages Database– Game Message **10**
Table 5 Log Index – Header **11**
Table 6 Log Index – Tag Entry **11**
Table 7 Log Index – Key-Frame Entry **12**

 Copyright © 2006 by Alion Science and Technology

1 Overview

After Action Review (AAR) is a verification and validation system, heavily utilized by simulation and training applications. It plays a vital role in determining the success or failure of a simulation. AAR supports multiple levels of mission debriefing and analysis, thereby providing the necessary evaluations of a particular training event.

Delta3D is an open source 3D game engine designed to help commercial, academic, and governmental organizations create training and education applications. Therefore, it is critical for the Delta3D game engine to support, at its core, an After Action Review system. This implies a necessity for creating a generic software infrastructure that can be used for gaming as well as training and simulation applications.

The Software Design Document (SRD) for After Action Review is broken into two major systems: 1) logging and playback; and 2) task tracking. This particular SDD discusses the logging and playback behavior. Task tracking is defined in Volume 2.

1.1 Logging

At the core of any AAR system is the ability to record (log), and playback a particular simulation or training exercise. This mechanism allows a player or instructor official to review the results and see what went right or wrong. In addition, entertainment based games may wish to capture "video" for creating demos of their games.

The logging capability provides the ability to capture messages to a log file ('record') and then play them back ('playback'). Tags may be inserted at any point in the log file to designate important points of interest. Key-frames may also be inserted into the log file serving as an absolute position within the simulation. Key-frames provide applications with the ability to jump around the log file without waiting for the playback to reach a certain point.

In order to illustrate the log file concept, consider a DVD player. A DVD player has mechanisms for fast-forward, pausing, and playing. This is similar to the playback facilities offered by the logging component with one exception. Real-time rewind is not possible in this implementation. A user my only go backwards through the recorded log if a key-frame resides at some point in time prior to the current time marker. Jumping to a keyfame is very similar to selecting a chapter from the DVD menu. Selecting a chapter starts the DVD at that point in time.

The primary goal of this design is to create a functional architecture. Although there is a test application (testAAR) which fully demonstrates this capability, it is not the primary intent of this design. To see a complete, working example of this design with heads up display (HUD), see the testAAR test application. Outside of that app, there is no default UI for this behavior.

 Copyright © 2006 by Alion Science and Technology

2 Component Design

The components comprising the AAR system are written using a component or aggregate based design strategy. This is used to ensure maximum flexibility and longevity of the system. In doing so, the AAR components fit naturally within the existing Game Manager infrastructure available within Delta3D. The Game Manager tracks and manages game related components, game actors, and routes game messages to and from them. Therefore, a Game Manager component serves as an excellent entry point for performing AAR related actions. For a detailed design pertaining to the Game Manager and the Game Manager components, see the Game Manager Design Document.

In order to allow logging in a networked client-server environment, the Delta3D AAR logging behavior is broken into two separate Game Manager components: 1) the log controller component and 2) the server logger component. The server logger component does most of the work whereas the log controller component is how an application will send commands and check the status of logging.

This section discusses the designs for these two new Game Manager components. Figure 1 illustrates a high-level overview of the logging component(s). Note, although the presence of a network is depicted in the diagram, it is not required. The *ServerLoggerComponent* and *LogController* can (and often does) reside on one machine.

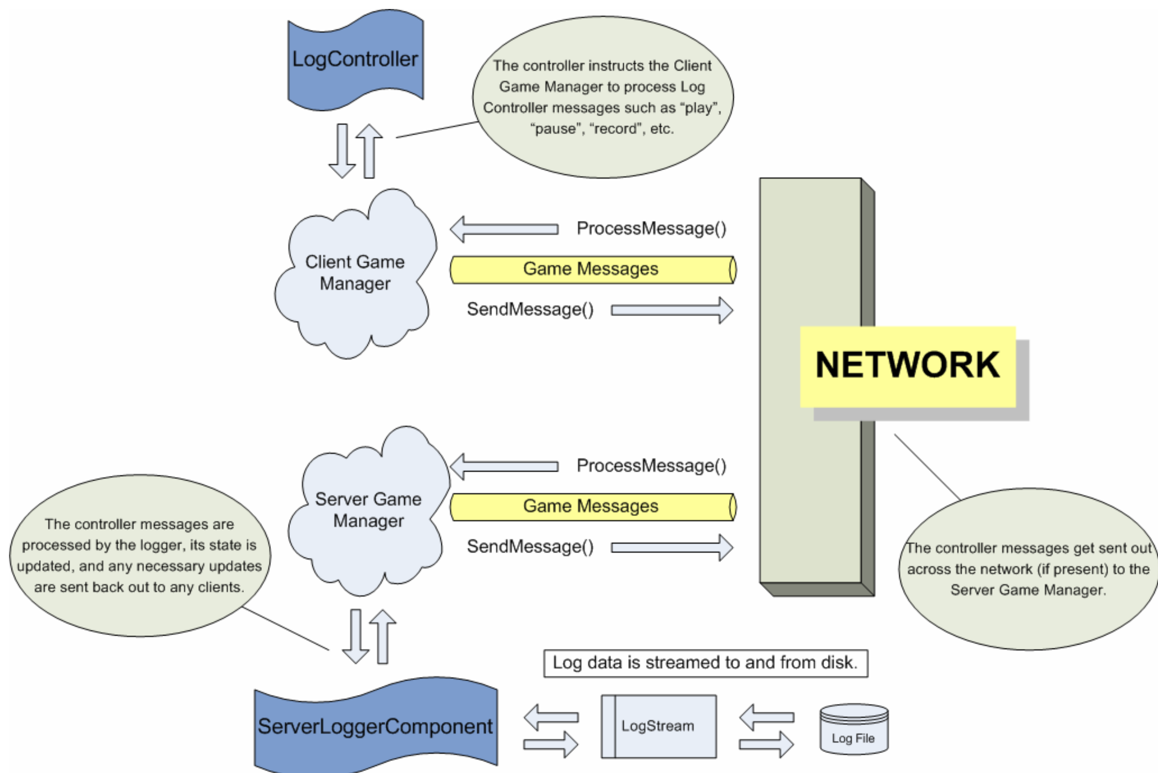

Figure 1 Logger Components Overview

 Copyright © 2006 by Alion Science and Technology

2.1 Server Logger Component

The first component is the *ServerLoggerComponent.* It serves primarily as a "back-end" component; it does all of the work. This component is the primary entry point to the log files or log stream interface discussed in Section 2.1.2.2. Although you will need to add one of these to your application, you should never be interacting directly with this component. The ServerLoggerComponent is driven entirely by messages that it receives from the Game Manager.

2.1.1 Class Diagram

The *ServerLoggerComponent* is depicted in the following class diagram.

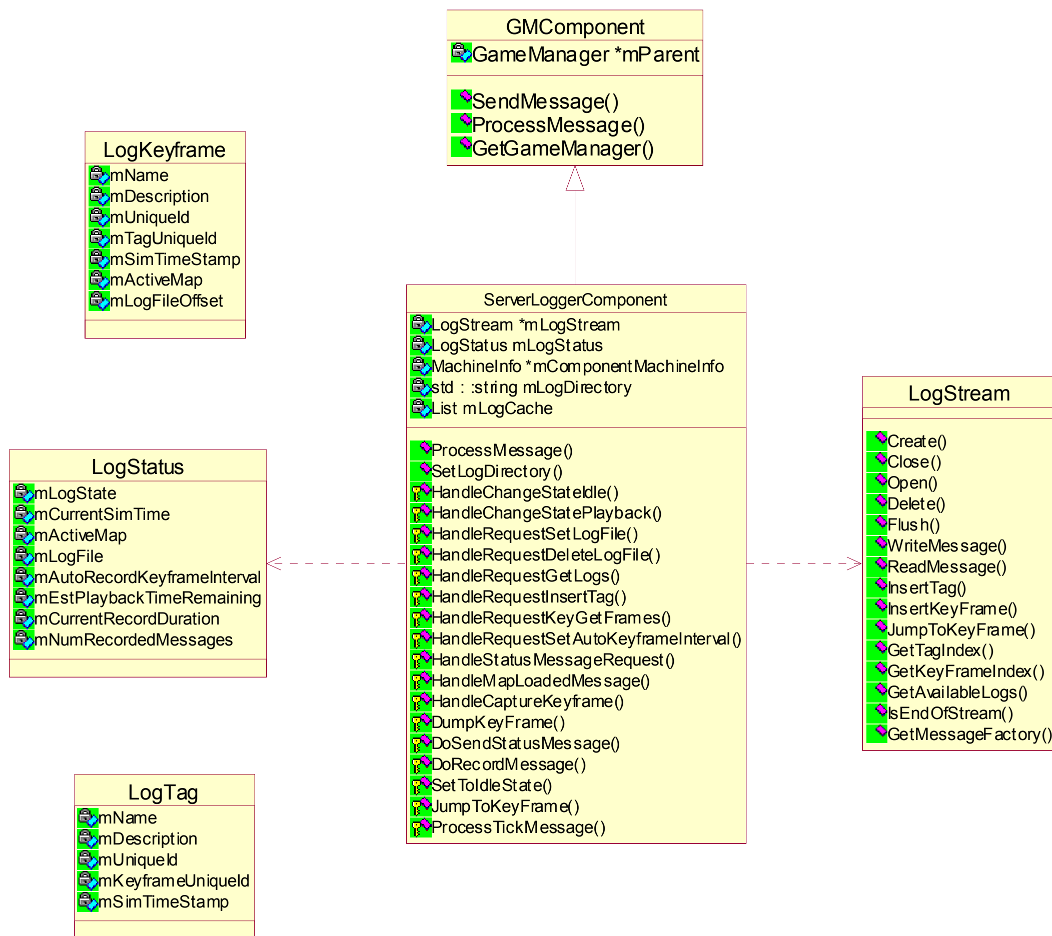

Figure 2 ServerLoggerComponent Class Diagram

2.1.2 Class Descriptions

The *ServerLoggerComponent* class diagram is described by the following classes. Note, the *LogStream* is an abstract interface class. This will allow developers to create custom storing mechanisms. The default implementation provides a binary log stream. See Section 2.3 for a discussion pertaining to the log file and binary log stream designs.

 Copyright © 2006 by Alion Science and Technology

2.1.2.1 ServerLoggerComponent

This class is the primary interface to the actual log stream and underlying log data. Therefore, when referring to the Delta3D logger, typically, this class will be the primary focus. The logger design is centered on message tracking. Therefore, the server logger component intercepts all game messages going through the Game Manager. As noted in the class diagram it is a subclass of *GMComponent*. Therefore, it has access to all game messages that are sent in the system.

In addition to recording all messages in the system, the *ServerLoggerComponent* also responds to messages specific to logging and playback. Section 3 contains a table of messages specific to the logging capabilities. By responding to these messages, the *ServerLoggerComponent* exposes the ability to both record and playback a log file. Tags and key-frames are also inserted into the log stream via responses to these messages.

In a client/server environment, this component should typically be added to the server; however, the design does not prevent a user or application from adding this component to a client. In either case, there should be one and only one *ServerLoggerComponent* present in an application or simulation. The default binary stream will store the resulting log files on the machine where the *ServerLoggerComponent* is instanced.

2.1.2.2 LogStream

This class is a pure virtual interface to a log file(s). It is responsible for streaming data to and from the actual log file(s). This interface is used by the *ServerLoggerComponent*. Again, this is only an interface class. Any data buffers, message queues, etc. should be on subclasses. The class diagram for *LogStream* depicts the methods exposed by the interface.

2.1.2.3 LogTag

This is a simple data class corresponding to a tag entry. It has several members for tracking bookkeeping information about tags such as the tag’s name, description, and time-stamp. Time-stamps are double precision decimal values representing the number of seconds since the beginning of the simulation (sim-time) when the tag was generated.

2.1.2.4 LogKeyframe

The *LogKeyframe* stores information about a key-frame and is very similar, in concept, to the *LogTag* data class. *LogKeyframe* holds the meta-data for a particular point-in-time system dump (‘keyframe’). Most data elements are simple or self-explanatory. However, there are two attributes worth discussing. First, the *LogKeyframe* class contains a string referring to the current map. This allows the key-frames to capture the current state of the actors in the system as well as the environment (i.e. the map) in which they reside. Second, the log key-frame has an offset into the log file. This offset is used to instruct a particular *LogStream* implementation where the start of the actual key-frame data exists within the stream. Although the offset is stored in the keyframe, it should be considered low-level information and your applications should not attempt to set or get it directly.

2.1.2.5 LogStatus

The *LogStatus* class is used to track the current state of the logger component. It maintains tracking information about how many messages were recorded, the amount of remaining time when in playback, etc. This class is sent to the *LogController* whenever a status request is made and whenever a request is fulfilled or rejected. This class is the primary way that the *LogController* and the *ServerLoggerComponent* keep in sync. This class indicates the current state of the server logger: IDLE, PLAY, or RECORD. Certain requests are only valid in particular states.

2.2 LogController Component

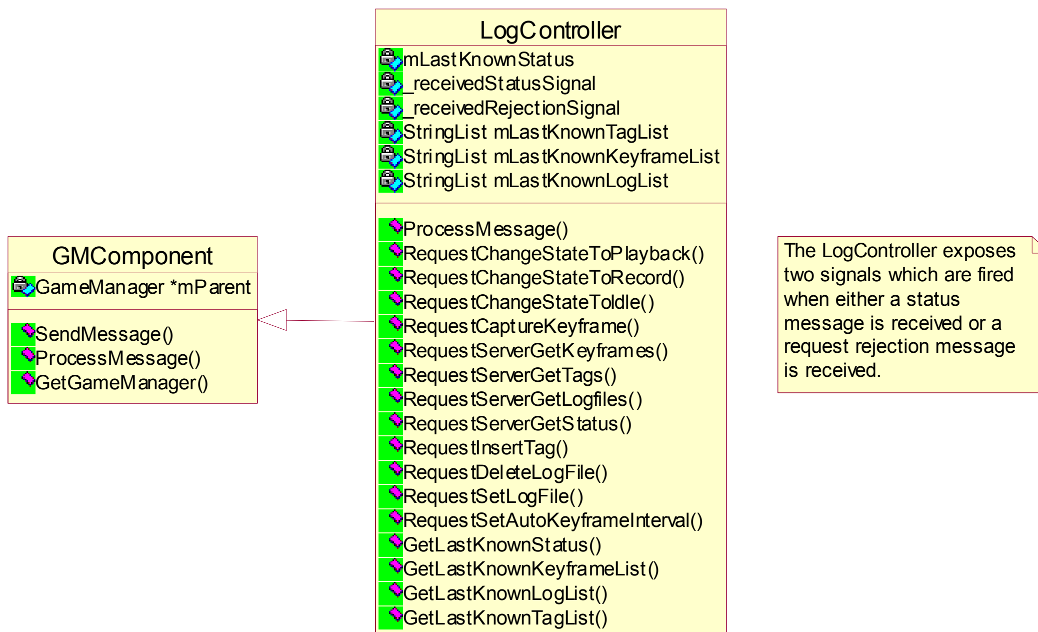

Figure 3 LogController Class Diagram

The second component is the *LogController*. This is the component that users should use to control the logging process. It serves as the driver for the *ServerLoggerComponent* and is responsible for keeping itself updated and synchronized with it. Therefore, users should call methods on this component which in turn, generate the appropriate messages for communicating a particular action to the logging system.

The *LogController* does exactly what its name implies; it controls the logging and replay process of the ServerLoggerComponent. Again, this class is a subclass of *GMComponent* so it can both send and process game messages. Most of the behaviors on this class are simply wrappers that send a request to the *ServerLoggerComponent* to perform an action. For example, to set a new log file, call the *RequestSetLogFile()* method. To start recording to the log file, call *RequestChangeStateToRecord()*. In actuality, these methods

aren't doing the real work; they are causing game messages to be sent via the Game Manager to the ServerLoggerComponent. See the Logger Messages section (Table 1 Logger Messages) for a complete list of messages.

In addition to exposing an interface for communicating to the *ServerLoggerComponent,* the *LogController* also listens for responses from the server and reports them to the end user. This is accomplished through the use of a signal/slot construct. There are several signals exposed by the *LogController* including: SignalReceivedStatus(), SignalReceivedTags(), SignalReceivedKeyframes(), and SignalReceivedRejection(). These are each called at appropriate times and should be reasonably self-explanatory. The rejection signal is invoked when an error/rejection messages is generated from the *ServerLoggerComponent*.

A *LogController* component can exist on any of the nodes (client or server) within the simulation. Therefore, the design allows multiple people (instructors) to control the flow of the logging process and insert tags and/or key-frames. By using game messages as the method of communication between the controller and the actual log, we ensure seamless integration of networked and non-networked interfaces.

2.3 BinaryLogStream

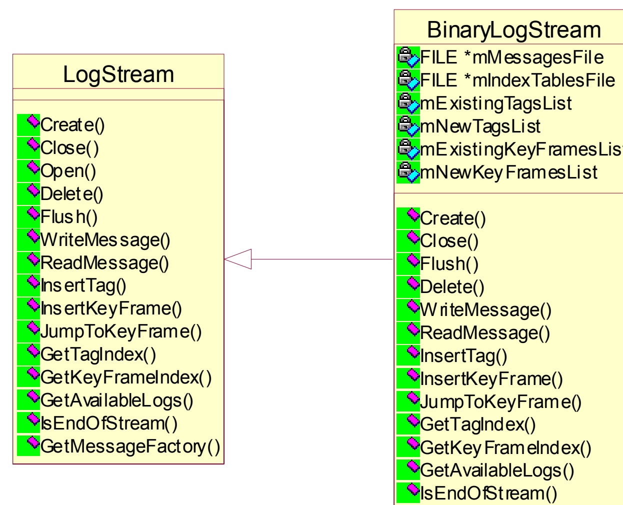

Figure 4 BinaryLogStream Class Diagram

The *BinaryLogStream* is a particular implementation of the LogStream that uses a binary data storage format. It is the default format exposed by this design. The details of the format can be found in Section 4. The *BinaryLogStream* is a subclass of *LogStream* and implements the required streaming operations based on the log file format below.

The *BinaryLogStream* also manages a list of keyframes and a list of tags. The existing tag and keyframe lists represent what was loaded from the log file. The new tag and keyframe lists contain tags and keyframes that were added during the record or playback

phase. Before closing the log stream, implementations should flush these lists to the log file(s) in order to correctly reflect any additional tags or keyframes that may have been inserted.

A potential future improvement would be to support a double buffered mechanism for reading data from and writing data to the log file. By doing so, the *BinaryLogStream* could support a threaded architecture where a read/write thread is processing one buffer, while the stream is filling the second. The current implementation performs reads/writes synchronously (ie. blocking).

3 Messages

This section contains tables for specialized messages deployed within the Delta3D AAR subsystem. These messages are game messages and the tables below follow the same conventions used in the Game Manager SDD.

3.1 Logger Messages

Enumeration	Name	Category	Description	Data
LOG_REQ_ CHANGESTATE_PLAYBACK	"Logger - Change State to Playback"	Request	Requests that the ServerLoggerComponent change to playback mode. Plays the current log file set with the "Set Log" message. (Only from IDLE)	Nothing
LOG_REQ_ CHANGESTATE_RECORD	"Logger - Change State to Record"	Request	Requests that the ServerLoggerComponent change to record mode. (Only from IDLE)	Nothing
LOG_REQ_ CHANGESTATE_IDLE	"Logger - Change State to Idle"	Request	Requests that the ServerLoggerComponent change to idle mode. Ends record/playback and closes stream. (ANY STATE)	Nothing
LOG_REQ_ CAPTURE_KEYFRAME	"Logger - Capture New Keyframe"	Request	Asks the ServerLoggerComponent to snap a key-frame (Default only supports RECORD)	LogKeyframe
LOG_REQ_JUMP_TO_KEYFRAME	"Logger - Jump To Keyframe"	Request	Asks the ServerLoggerComponent to jump to a keyframe. (Only from PLAYBACK)	LogKeyframe
LOG_REQ_GET_KEYFRAMES	"Logger - Get List of Keyframes"	Request	Asks the ServerLoggerComponent for the current list of keyframes. (RECORD or PLAYBACK)	Nothing
LOG_REQ_GET_LOGFILES	"Logger - Get List of Log Files"	Request	Asks the ServerLoggerComponent for the current list of logs. (ANY STATE)	Nothing
LOG_REQ_GET_TAGS	"Logger - Get List of Tags"	Request	Asks the ServerLoggerComponent for the tags currently in the log. (ANY STATE)	Nothing

Enumeration	Name	Category	Description	Data
LOG_REQ_GET_STATUS	"Logger - Get Status"	Request	Asks the ServerLoggerComponent for a status update. (ANY STATE)	Nothing
LOG_REQ_INSERT_TAG	"Logger - Add Tag"	Request	Requests that the ServerLoggerComponent insert a tag at the current log stream position. (Default only supports RECORD)	LogTag
LOG_REQ_DELETE_TAG	"Logger - Delete Log"	Request	Attempts to delete a tag from ServerLoggerComponent's current log stream. (Default only supports RECORD)	LogTag
LOG_REQ_SET_LOGFILE	"Logger - Set Current Logfile"	Request	Attempts to set the current log on the ServerLoggerComponent. (Only in IDLE)	Log name
LOG_REQ_SET_ AUTOKEYFRAMEINTERVAL	"Logger - Set Auto Keyframe Interval"	Request	Request that the ServerLoggerComponent change its auto keyframe capture interval. (ANY STATE – Used by record)	Interval in seconds (double)
LOG_INFO_KEYFRAMES	"Logger - Info - Keyframes"	Info	Contains a list of keyframes found in the current log. (ANY STATE)	List of LogKeyframe s
LOG_INFO_LOGFILES	"Logger - Info - Logfiles"	Info	Contains a list of log files located by the ServerLoggerComponent (ANY STATE)	List of strings representing log names.
LOG_INFO_TAGS	"Logger - Info - Tags"	Info	Contains a list of tags found in the current log. (ANY STATE)	List of Log Tags.
LOG_INFO_STATUS	"Logger - Info - Status"	Info	Contains a snapshot of the current state of the ServerLoggerComponent (ANY STATE)	LogStatus
LOG_COMMAND_BEGIN_ LOADKEYFRAME_TRANS	"Logger - Server Command - Begin Load Keyframe Transaction"	Command	Informs the world that a keyframe is about to be loaded. (PLAYBACK)	Nothing
LOG_COMMAND_END_ LOADKEYFRAME_TRANS	"Logger - Server Command - End Load Keyframe Transaction"	Command	Informs the world that a keyframe is done being loaded. (PLAYBACK)	Success flag Failure reason

Table 1 Logger Messages

4 Log File Format

As stated in Section 2.1.2, the *ServerLoggerComponent* uses the *LogStream* interface to stream data to and from the underlying log file. The *LogStream* abstract class can be extended to support a human-readable file format such as XML; however, due to performance considerations, only a binary log stream was considered for this design. This section describes the format the logger component uses when writing log files in binary form.

The binary log file format is designed to be optimal for both storage and frequent read/write operations. The format is represented using two separate files. The first file contains a database of messages that are (de)serialized directly to/from the file (.DLM extension). The second file is an index into the message database file (.DLI extension). The second file also contains an interleaved list of tags and keyframes. The tag and keyframe entries may contain offsets into the message database file to signal the start of any relevant data mapping to that particular entry. Both files are named the same with the exception of their extension. For example, if a user wants to create a log named "foobar", the two files would be "foobar.dli" and "foobar.dlm".

Game Messages (see Game Manager SDD) supports several data types; therefore the log file must support those data types as well. There are, however, a few data types this format uses that are not within the scope of the Game Message structure. The following table lists the basic data types utilized by this log file format. These data types are referenced throughout this section.

Type	Description
ubyte	unsigned byte
uint	4-byte unsigned integer
int	4-byte signed integer
ushort	2-byte unsigned integer
string[n]	String of *n* ASCII bytes, not necessarily null-terminated. If no length is specified, the length of the string is assumed to be derived from another source.
Double	8-byte decimal number.

Table 2 Log File Format – Basic Data Types

4.1 DLM File Structure (Delta Logger Messages)

The DLM file structure begins with a header followed by a list of messages. The following two subsections describe the contents of the file header and message list. This file format is specifically designed for the Delta3D engine's messaging infrastructure. The file stores the information needed to re-create new message objects. The message object itself knows how to serialize and de-serialize itself to and from a byte stream. However, third party applications could document all types of messages stored in the log such that an external file parser could be implemented. This log file format directly relies on a message's ability to handle its own data serialization.

4.1.1 Header

All DLM log files must begin with this specific header or it will be considered invalid.

Type	Description
string[10]	A string identifier or "magic number". This should be equal to "GMLOGMSGDB". It is not null-terminated.
ubyte	Major version number. Should equal 1 via this document.

 Copyright © 2006 by Alion Science and Technology

ubyte	Minor version number. Should equal 0 via this document.
double	Duration of the simulation time recorded by this log. Note that the actual starting simulation time of the recording can be found from the first LogKeyframe.
ushort	Size of the index file name.
string[]	Index file name. This length of this string should be equal to the specified size in the previous field.

Table 3 Messages Database – Header

4.1.2 Game Message

The rest of the DLM file is basically a continuous stream of game messages. Each game message has the following format.

Type	Description
ubyte	Element descriptor id. For messages this is always equal to 0
ushort	Game Message ID. The ID maps to an enumeration which corresponds to a particular type of Game Message. The reader uses this ID to create the message which then extracts its data from the byte stream. Note that you cannot replay a log file if the message ID's have changed. Message ID's are defined when messages are registered with the MessageFactory – see messagetype.cpp.
double	Timestamp in simulation time when this message was written.
uint	The size in bytes of the data corresponding to the message. The reader should read this data into a buffer and parse the message from memory.
MessageData	Message specific data. It is the messages responsibility to parse the byte stream to extract the information.

Table 4 Messages Database– Game Message

4.2 DLI File Structure (Delta Logger Index)

The DLI file contains the extra information associated with a recording such as tags and keyframes. Since keyframes and tags need to be referenced out of sequence, we can't store them in the main body of messages. This file is comprised of two sections: the file header and an interleaved list of tags and keyframes. These are denoted by their DEID or data element identifier.

4.2.1 Header

All DLI log files must begin with the following header. If the file does not begin with this header, it will be considered invalid.

Type	Description

 Copyright © 2006 by Alion Science and Technology

string[13]	A string identifier or “magic number”. This should be equal to “GMLOGINDEXTAB”. It is not null-terminated.
ubyte	Major version number. Should equal 1 via this document.
ubyte	Minor version number. Should equal 0 via this document.
ushort	Size of the message database file name.
string[]	Message database file name. The length of this string should be equal to the specified size in the previous field.

Table 5 Log Index – Header

4.2.2 Tags and Keyframes

This section describes the tags and key-frames table located in the index file. There are no preset limits to the number of keyframes and/or tags located in the file. Parsers should continue reading from the file until either an error or end of file is encountered using the DEID value to determine whether to parse a key-frame or a tag.

Note – custom LogStreams may store extra information in this part of the index file.

4.2.2.1 Tag

Tags are read from and written to the index file using the *DataStream* interface discussed in the Game Manager SDD. The following table explains the tag data stored in the index file.

Type	Description
ubyte	DEID value. This should equal 1.
uint	Size of the tag data in bytes.
TagData	Stream of bytes which contain the following data in the following order: tag name (**string**), tag description (**string**), tag time stamp (**double**), and tag UniqueId. This data should be parsed using the existing *DataStream* interface.

Table 6 Log Index – Tag Entry

4.2.2.2 Key-Frame

Key-Frames are read from and written to the index file using the DataStream interface discussed in the Game Manager SDD. A key-frame is a snapshot of the entire current state of the simulation; therefore, there is a lot of data in a key-frame. Most of this data is stored in the index file’s corresponding DLM file in the form of ActorUpdateMessages. Since keyframes are used to jump around (forward or backward) in the log stream, we need some header information for each one. So, for each keyframe in the main file, we need an entry like this:

Type	Description
ubyte	DEID value. This should equal 2.
uint	Size of the key-frame data in bytes.
Key-	Stream of bytes which contain the following data in the following order:

Frame Data	key-frame name (**string**), key-frame description (**string**), key-frame time stamp (**double**), key-frame UniqueId, active map (**string**), and log file offset (**long**). This data should be parsed using the existing *DataStream* interface.

Table 7 Log Index – Key-Frame Entry

5 Demonstration Application(s) and Unit Tests

The demonstration application shows all aspects of this design. The demo application consists of a logging portion (recording) and a playback portion. (playback). More specifically, the demonstration application should do the following:

- Display a simple walk-thru scenario providing keyboard commands to record and playback their actions.
- Should allow the user to drop objects around the scene and capture those actions.
- Application supports both tags and keyframes providing feedback to the user with regards to the current tags and keyframes.
- Provide a simple representation of a player.
- Allow the user to insert tags during playback.
- Allow the user to pause and resume the playback cycle.
- Provide status text feedback to the user.
- Allow the user to adjust the rate at which the scenario is both recorded and viewed during playback.

Note, in addition to the above list of demonstration points, all aspects of this design are thoroughly covered with unit tests.

6 Notes

- Future versions of the logger file format should probably use some form of compression. Using the gzip file format would be fairly easy to implement if future versions of the logger decide to go with a compressed format.
- Future versions might want to support some mechanism for capturing a screenshot when saving a key-frame.
- Future versions could support some sort of permissions system on the LogController. For example, if there are multiple controllers how do we keep them from stepping on each other's toes?

Appendix A: Glossary

Acronym	Definition
AAR	After Action Review
API	Application Programming Interface
DLM	Delta Logger Messages file
DLI	Delta Logger Index file
DAL	Dynamic Actor Layer
GM	Game Manager
JNTC	Joint National Training Capability
SDD	Software Design Document
SRD	System Requirements Document
STAGE	Simulation, Training, and Game Editor
UI	User Interface

 Copyright © 2006 by Alion Science and Technology