



DI-Guy SDK

Developers Guide



VT MÄK
A company of VT Systems



DI-Guy SDK

Developers Guide

Copyright © 2014 VT MÄK

All rights Reserved. Printed in the United States. No part of this document may be copied or reproduced in any form without prior written consent of VT MÄK.

VR-Exchange™, VR-TheWorld™, VR-Vantage™, DI-Guy™, and DI-Guy Scenario™ are trademarks of VT MÄK. MÄK Technologies®, VR-Forces®, RTIsby®, B-HAVE®, and VR-Link® are registered trademarks of VT MÄK.

3ds Max® is a registered trademark of Autodesk, Inc.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Vega Prime and OpenFlight are registered trademarks of Presagis.

Visual C++ and Direct3D are registered trademarks of Microsoft Corp.

FLEXIm is the registered trademark of Flexera.

COLLADA is a trademark of Sony Computer Entertainment, Inc.

Certain models were provided by CG2, Presagis, Antycip, Engineering and Computer Simulations, Inc., and Viewpoint DataLabs International.

E-Town™ Building Library models are from CGSD Corporation, © 1999.

libjpeg - this software is based in part on the work of the Independent JPEG Group

libtiff © 1988-1997 Sam Leffler, Silicon Graphics, Inc. www.libtiff.org

Ogg Vorbis © 2002 Xiph.org Foundation www.xiph.org

OpenAL is a trademark of Creative Labs, Inc.

Perl - © 1989-1999, Larry Wall www.perl.org, www.activestate.com

Qt © 2005-2007 Trolltech ASA www.trolltech.com

zlib © 1995-2005 Jean-loup Gailly and Mark Adler www.zlib.net

pthread sourceware.org/pthreads-win32

Qscintilla © 2008 Riverbank Computing Limited www.riverbankcomputing.co.uk

FaceFX. ©2002-2013, OC3 Entertainment, Inc

All other trademarks are owned by their respective companies.

For third-party license information, please see “Third Party Licenses,” on page xiii.

VT MÄK
150 Cambridge Park Drive, 3rd Floor
Cambridge, MA 02140 USA

Voice: 617-876-8085

Fax: 617-876-9208

info@mak.com

www.mak.com

Revision DIG-13.0-7-140623



Contents

Preface

How This Manual is Organized	vii
Documentation	viii
Derivative Work	viii
MÄK Products	ix
How to Contact Us	xi
Document Conventions	xii
DI-Guy Conventions	xiii
Mouse Button Naming Conventions.....	xiii
Third Party Licenses	xiii
Boost License.....	xiii
libXML and libICONV	xiv
Lua.....	xiv

Chapter 1. Introduction to the DI-Guy SDK

1.1. DI-Guy SDK Use Cases	1-2
1.1.1. Embed Scenarios Created with DI-Guy Scenario	1-3
1.1.2. Create and Control Characters in Real-time Using the DI-Guy API	1-4
1.1.3. Network with Other Simulators (DIS or HLA)	1-5
1.2. Querying and Setting Actions in a Pipeline Setting	1-7
1.2.1. Host Side	1-7
1.2.2. IG Side	1-8

Chapter 2. DI-Guy SDK Concepts

2.1. How DI-Guy Works	2-2
2.1.1. The Character Viewer	2-2
2.1.2. DI-Guy Skeletal Structure	2-3
2.1.3. The Basic DI-Guy Function Calls	2-3
2.1.4. A Brief DI-Guy Class Overview	2-6
2.1.5. Best Practices for Using DI-Guy	2-7
2.2. Working with the <i>diguyScenario</i> and <i>diguyCharacter</i> Classes	2-9
2.2.1. The <i>diguyScenario</i> Class	2-9
2.2.2. The <i>diguyCharacter</i> Class	2-14
2.3. The DI-Guy Graphics API	2-23

Chapter 3. The Motion Pipeline

3.1. The Motion Pipeline	3-2
3.1.1. The Initial Stage	3-2
3.1.2. Gazing and Pointing	3-4
3.1.3. Aiming	3-5
3.1.4. User Pose Overrides	3-5
3.1.5. Gestures	3-6
3.1.6. Head Nodding and Shaking	3-6

Chapter 4. Maximizing DI-Guy Performance

4.1. Techniques for Maximizing DI-Guy Performance	4-2
4.1.1. Using the Load Manager	4-4
4.1.2. Using DI-Guy as Part of a Networked Image Generator	4-5
4.1.3. Notes on Compressed Geometry, Textures, and Preloading ...	4-6

Chapter 5. Networking with DI-Guy

5.1. Networking with DI-Guy	5-2
5.1.1. Extending DIS and HLA for High Fidelity Visual and Motion Correlation	5-2
5.2. DIS Networking Using the SDK	5-3
5.2.1. Controlling the Update Rate	5-4
5.2.2. How DI-Guy Custom PDUs are Defined	5-4
5.2.3. DI-Guy Entity Mapping in the Absence of Custom PDUs	5-6
5.3. HLA Networking	5-7
5.3.1. RTI Compliance	5-7
5.3.2. Setting up DI-Guy to use HLA	5-8
5.3.3. Enabling HLA Networking in DI-Guy	5-10
5.3.4. Controlling the Update Rate	5-11
5.3.5. DI-Guy Entity Mapping	5-11
5.3.6. RPR FOM Elements Used by DI-Guy	5-12
5.3.7. DI-Guy FOM Extensions for HLA	5-12
5.3.8. Troubleshooting Your HLA Application	5-13
5.4. Fire and Detonate Interactions in DI-Guy	5-13

5.5. Network Configuration Parameters for DI-Guy Scenario	5-15
5.6. Database Coordinates and DI-Guy Networking	5-18
5.6.1. Setting the Exercise Location in DI-Guy	5-19
5.6.2. UTM Database Coordination Translation	5-19
5.7. Controlling DI-Guy Scenario Remotely Using Custom PDUs	5-19
Chapter 6. Writing Plug-ins for DI-Guy Scenario	
6.1. Writing Plug-Ins	6-2
6.1.1. Supported Plug-in Functions	6-2
Chapter 7. DI-Guy Author	
7.1. Introduction	7-2
7.2. How DI-Guy Author Works	7-2
7.2.1. How the End User Interacts with Farscene IG	7-2
7.2.2. How DI-Guy Functionality was Added to the IG	7-3
7.3. IG Integration	7-5
7.4. DI-Guy Author Programming Reference	7-6
7.4.1. DI-Guy Author Interface Class	7-6
7.4.2. 3D Visual Classes	7-7
Chapter 8. Compiling and Linking DI-Guy	
8.1. Compiling and Linking DI-Guy on Windows	8-2
8.1.1. Release and Debug Versions of DI-Guy Libraries	8-2
8.1.2. Lua Script Modules	8-3
8.1.3. Expressive Faces Module for OpenGL	8-3
8.1.4. Programming Examples	8-3
8.2. Compiling and Linking DI-Guy on Linux	8-3
8.2.1. Programming Examples	8-4
Chapter 9. Programming Examples	
9.1. DI-Guy Programming Examples	9-2
9.1.1. OpenGL Examples	9-3
Appendix A. Improving Loadtime Performance	
A.1. Creating a Custom Configuration File	A-2
A.1.1. Using Include Statements in DI-Guy Configuration Files	A-3

Index



Preface

This manual is written for persons who will install DI-Guy® or use the DI-Guy SDK. The manual assumes that you are familiar with your operating system and that you know how to work in your computer's graphical window environment.

Please see release documentation for information about changes and updates to DI-Guy since this manual went to press.

How This Manual is Organized

The chapters are organized as follows:

Chapter 1, *Introduction to the DI-Guy SDK*, describes common ways that customers use DI-Guy.

Chapter 2, *DI-Guy SDK Concepts*, covers basic concepts that will help you understand the rest of this manual.

Chapter 3, *The Motion Pipeline*, explains how DI-Guy processes motions.

Chapter 4, *Maximizing DI-Guy Performance*, describes techniques you can use to maximize DI-Guy performance.

Chapter 5, *Networking with DI-Guy*, describes DI-Guy support for DIS and HLA.

Chapter 6, *Writing Plug-ins for DI-Guy Scenario*, explains how to use the SDK to write plug-ins for use in DI-Guy Scenario.

Chapter 7, *DI-Guy Author*, explains how to use the SDK to embed DI-Guy Scenario in an application.

Chapter 8, *Compiling and Linking DI-Guy*, explains how to build and compile applications using the SDK..

Chapter 9, *Programming Examples*, briefly describes the examples included with the SDK.

Documentation

DI-Guy comes with the following documentation:

- ♦ *DI-Guy SDK Developers Guide*. Explains how to program using the DI-Guy SDK.
- ♦ *DI-Guy Scenario Users Guide*. Explains how to use DI-Guy Scenario to create scenarios with realistic human activity.
- ♦ *DI-Guy Content Reference Manual*. Explains how to customize DI-Guy characters and add new character models.
- ♦ *DI-Guy AI Users Guide*. Explains how to use DI-Guy AI to add intelligent behavior to DI-Guy characters.
- ♦ *DI-Guy Motion Editor Users Guide*. Explains how to import, edit, and customize DI-Guy motions.
- ♦ *DI-Guy Expressive Faces Users Guide*. Provides a brief introduction to using the Expressive Faces module.
- ♦ *DI-Guy Master Index*. A PDF master index to the PDF manuals.
- ♦ DI-Guy Master Index. An HTML page that links to all documentation.
- ♦ Class documentation. HTML documentation describing all classes and functions.

These documents are available from the start menu under DI-Guy, or by browsing to *./doc*. Of particular interest is the DI-Guy Documentation Master Index, which lets you quickly access all the documents above along with additional technical information about DI-Guy.

Derivative Work

Any DI-Guy content (model, texture, motion, and so on) that you modify or copy is a derivative work that is protected under copyright law by the same rules that apply to content delivered with DI-Guy. This derivative content may be used only on computers that have a valid DI-Guy license. Please contact VT MÄK to discuss licensing for other uses of derivative models.

MÄK Products

DI-Guy is a member of the VT MÄK line of software products designed to streamline the process of developing and using networked simulated environments. The VT MÄK product line includes the following:

- ♦ **VR-Link® Network Toolkit.** VR-Link is an object-oriented library of C++ functions and definitions that implement the High Level Architecture (HLA) and the Distributed Interactive Simulation (DIS) protocol. VR-Link has built-in support for the RPR FOM and allows you to map to other FOMs. This library minimizes the time and effort required to build and maintain new HLA or DIS-compliant applications, and to integrate such compliance into existing applications.

VR-Link includes a set of sample debugging applications and their source code. The source code serves as an example of how to use the VR-Link Toolkit to write applications. The executables provide valuable debugging services such as generating a predictable stream of HLA or DIS messages, and displaying the contents of messages transmitted on the network.

- ♦ **MÄK RTI.** An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK RTI is optimized for high performance. It has an API, RTIsPy®, that allows you to extend the RTI using plug-in modules. It also has a graphical user interface (the RTI Assistant) that helps users with configuration tasks and managing federates and federations.
- ♦ **VR-Forces®.** VR-Forces is a computer generated forces application and toolkit. It provides an application with a GUI, that gives you a 2D and 3D views of a simulated environment.

You can create and view local entities, aggregate them into hierarchical units, assign tasks, set state parameters, and create plans that have tasks, set statements, and conditional statements. VR-Forces also functions as a plan view display for viewing remote entities taking part in an exercise. Using the toolkit, you can extend the VR-Forces application or create your own application for use with another user interface.

- ♦ **VR-Vantage™.** VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes three end-user applications (VR-Vantage Stealth, VR-Vantage PVD, and VR-Vantage IG) and the VR-Vantage Toolkit.
 - VR-Vantage Stealth displays a realistic, 3D view of your virtual world, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world. You can move your viewpoint to any location in the 3D world and can attach it to entities so that it moves as they do.
 - VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.
 - VR-Vantage PVD provides a 2D plan view display. It gives you the big picture of the simulated world.

- The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.
- MÄK Data Logger. The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.
- VR-inTerra. VR-inTerra is a C++ API for adding terrain agility to applications. It can load, page, or stream terrain from a wide variety of formats or sources into a single, consistent run-time representation, consisting of a collision graph and vector network.
- DI-Guy. The DI-Guy product line is a set of software tools for real-time human visualization, simulation, and artificial intelligence. Every DI-Guy software offering comes with thousands of ready-to-use characters, appearances, and motions. DI-Guy enables the easy creation of crowds and individuals who are terrain aware, autonomous, and react intelligently to ongoing events. Save time, money and create outstanding simulations with DI-Guy. The DI-Guy product line includes the following products:
 - The DI-Guy SDK. Embed the DI-Guy library in your real-time application and populate your world with lifelike human characters.
 - DI-Guy Scenario. Author and visualize human performances in a rich, user-friendly graphical environment. Use DI-Guy Scenario as an end visualization application or save scenarios and load them into your DI-Guy SDK enabled application.
 - DI-Guy AI. Generate crowds of autonomous characters to quickly populate your worlds with hundreds and thousands of terrain-aware, collision avoiding DI-Guys. Used as a module on top of DI-Guy Scenario and DI-Guy SDK.
 - Expressive Faces Module. Enable DI-Guy characters to have faces that display emotion, eyes that look in directions and blink, and lips that sync to sound files.

- DI-Guy Motion Editor. Create or customize motions to your particular needs in an easy-to-use graphical application.

How to Contact Us

For DI-Guy technical support, information about upgrades, and information about other MÄK products, you can contact us in the following ways:

Telephone

Call or fax us at:	Voice:	617-876-8085 (extension 3 for support)
	Fax:	617-876-9208

E-mail

Sales and upgrade information:	info@mak.com
Technical support:	support@mak.com
VR-Vantage support:	vrv-support@mak.com

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/get-licenses.html
Product version and platform information:	www.mak.com/support/ product-versions.html
For the free, unlicensed MÄK RTI:	www.mak.com/resources/bonus- material/cat_view/16-bonus-materials/ 24-mak-high-performance-rti.html
MÄK Community Forum:	www.mak.com/community-forum/ 1-forum.html

Post

Send postal correspondence to:	VT MÄK 150 Cambridge Park Drive, 3rd Floor Cambridge, MA, USA 02140
--------------------------------	---

When requesting support, please tell us the product you are using, the version, and the platform on which you are running.

Document Conventions

This manual uses the following typographic conventions:

Monospaced	Indicates commands or values you enter.
Monospaced Bold	Indicates a key on the keyboard.
<i>Monospaced Italic</i>	Indicates command variables that you replace with appropriate values.
Blue text	A hypertext link to another location in this manual or another manual in the documentation set.
{ }	Indicates required arguments.
[]	Indicates optional arguments.
	Separates options in a command where only one option may be chosen at a time.
()	In command syntax, indicates equivalent alternatives for a command-line option, for example, (-h --help).
/	Indicates a directory. Since MÄK products run on both Linux and Windows PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
<i>Italic</i>	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose File → Save .
	Click the icon to run a tutorial video in the default browser.
	Indicates supplemental or clarifying information.
	Indicates additional information that you must observe to ensure the success of a procedure or other task.

Directory names are preceded with dot and slash characters that show their position with respect to the DI-Guy home directory. For example, the directory *vrforces13/doc* appears in the text as *./doc*.

DI-Guy Conventions

Table 2-1 lists the conventions used for entity coordinates in DI-Guy documentation.

Table 2-1: Coordinate system conventions

Convention	Indicates
XYZ	X = forward, Y = to the left Z = up
Yaw, Roll, Pitch	Orientation is applied in the order YRP. Yaw = rz – orientation about the vertical axis Roll = rx – orientation about the fore-aft axis Pitch = ry – orientation nose down, nose up

Mouse Button Naming Conventions

An instruction to click the mouse button, refers to clicking the primary mouse button, usually the left button for right-handed mice and the right button for left-handed mice. The context-sensitive menu, also called a popup menu or right-click menu, refers to the menu displayed when you click the secondary mouse button, usually the right button on right-handed mice and the left button on left-handed mice.

Third Party Licenses

MÄK software products may use code from third parties. This section contains the license documentation required by these third parties.

Boost License

VR-Link, and all MÄK software that uses VR-Link uses some code that is distributed under the Boost License. All header files that contain Boost code are properly attributed. The Boost web site is: www.boost.org.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the “Software”) to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

libXML and libICONV

VR-Link and all MÄK software that uses VR-Link, links in libXML and libICONV. On some platforms the compiled libraries and header files are distributed with MÄK Products. MÄK has made no modifications to these libraries. For more information about these libraries please see the following web sites:

- The LGPL license is available at: <http://www.gnu.org/licenses/lgpl.html>.
- Information about IconV is at: <http://www.gnu.org/software/libiconv/>.
- Information about LibXML is at: <http://xmlsoft.org/>.

Lua

VR-Forces and DI-Guy use the Lua programming language (www.lua.org). Its license is as follows:

Copyright © 1994–2012 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



1. Introduction to the DI-Guy SDK

This describes how DI-Guy customers use the SDK.

DI-Guy SDK Use Cases.....	1-2
Embed Scenarios Created with DI-Guy Scenario	1-3
Create and Control Characters in Real-time Using the DI-Guy API.....	1-4
Network with Other Simulators (DIS or HLA)	1-5
Querying and Setting Actions in a Pipeline Setting.....	1-7
Host Side.....	1-7
IG Side.....	1-8

1.1. DI-Guy SDK Use Cases

This chapter provides three detailed examples of how you can use the DI-Guy SDK to simulate entity behavior:

- ♦ Create a scenario using DI-Guy Scenario (with DI-Guy AI) and then load the scenario into your DI-Guy-enabled simulation and play it. The high-level construction is performed off-line in DI-Guy Scenario, a tool designed to quickly and efficiently generate compelling human behavior. This is the recommended use case.
- ♦ Create and control DI-Guy characters directly in your simulation using low-level calls to the DI-Guy Action API. In this use case, you code all the high-level construction.
- ♦ Drive DI-Guy characters from a network. Examples include being at the end of a DIS/HLA pipe or being in a Host-IG environment. Here the high-level construction is performed somewhere else and piped in.

The DI-Guy SDK includes programming examples. The examples assume a Windows operating system using OpenGL. Source code for the examples is provided with the DI-Guy distribution. They are in *./programming_examples/*. In addition to OpenGL, examples are provided for DirectX and OpenSceneGraph (OSG) in the DI-Guy Graphics API, featuring open source implementation details. DI-Guy can render on just about any Windows or Linux rendering system imaginable.

All the OpenGL Windows examples should be looked at by all users, not just OpenGL users. Although the rendering environments can be very different, almost all your DI-Guy code will look the same.

1.1.1. Embed Scenarios Created with DI-Guy Scenario

The DI-Guy SDK can load and play scenarios created using DI-Guy Scenario. The *.dss* scenario file describes the scenario, the characters, behaviors, AI, decisions, and logic that play out over time. The characters are still accessible using the DI-Guy API, so you can both monitor and alter their behavior at runtime.

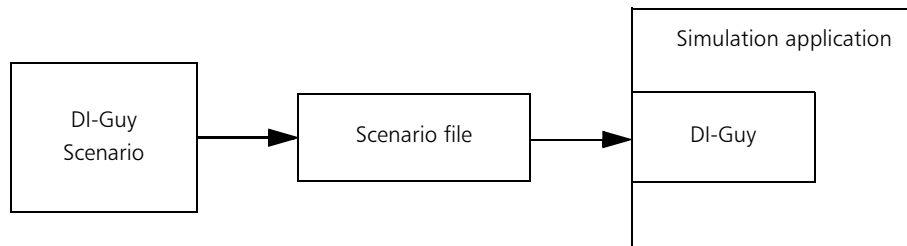


Figure 1-1. Using a DI-Guy scenario in an application

We think this is the most productive way to work with DI-Guy, because DI-Guy Scenario and DI-Guy AI are designed to let you quickly and efficiently create compelling human performances whether you are a programmer or not. The high-level process is as follows:

1. Use DI-Guy Scenario to create a scenario.
2. Load the scenario into your application through the DI-Guy API.
3. Play the scenario.

This method of adding humans is shown in the `simple_playback` example. This example is in `./programming_examples/diguy_ogl/simple_playback`.

(Details about programming examples can be found in Chapter 9, [Programming Examples](#).)

The following DI-Guy OpenGL example loads and then plays back a scenario file created with DI-Guy Scenario.

1. Initialize DI-Guy. This particular initialization works for OpenGL, but `diguy_graphics_api_initialize()` should be used for the DI-Guy Graphics API implementations.


```
diguy_ogl_initialize(NULL);
```
2. Create a DI-Guy scenario.


```
diguyScenario *scenario = diguy_create_scenario();
```
3. Load the *.dss* file into the scenario.


```
scenario->load(arg_dss_filename);
```
4. In your update loop, update the scenario with the current time in seconds.


```
scenario->update(t);
```

5. In your draw loop, draw the scenario. In scene graph style renderers, this step is not necessary.

```
scenario->draw();
```

6. 6. When finished, clean up the scenario and DI-Guy.

```
diguy_destroy_scenario(scenario);
diguy_deinitialize();
```

1.1.2. Create and Control Characters in Real-time Using the DI-Guy API

Another straight-forward way to use DI-Guy is to add DI-Guy characters to your application and then command their behaviors directly in real time using the DI-Guy API.

The “simple” example, creates a soldier character and then commands the character to perform a sequence of movements starting from a standing position. The source code for this example is in *./programming_examples/diguy_ogl/simple*. You can run the compiled example by executing *simple.exe* (Windows) or *simple* (Linux).

1. Initialize DI-Guy. In this example, we used the OpenGL version of DI-Guy.

```
diguy_ogl_initialize(NULL);
```

2. Create a DI-Guy scenario. You need at least one scenario to run DI-Guy.

```
diguyScenario* scenario = diguy_create_scenario();
```

3. Create a DI-Guy character of type soldier. You must specify the name, character type, and appearance of your character. You can specify body appearance, head appearance, and hand item. If you do not specify one of these, default values are used.

```
diguyCharacter* character1 = scenario->create_character(
    "Character 1", // name
    "soldier_07", // character type soldier
    NULL);        // default appearance
```

4. Set the initial position, orientation, and action for the character:

```
character1->set_position(0.0f, -3.0f, 0.0f);
character1->set_orientation(90.0f, 0.0f, 0.0f);
character1->force_action("kneel_ready");
```

5. You will update the scenario periodically using `scenario_update()`. Before calling `scenario_update()` in the update loop, make any calls to change the behavior of your character. Here `prev_t` is the value of `t` the last time through the loop. At 2 seconds the character is commanded to walk, at 8 seconds the characters walks in a crouch, at 14 seconds he gets on the ground and aims the weapon, and at 18 seconds he fires the gun in 1 second intervals. All transitions are handled automatically by the DI-Guy motion engine.

```
if ((t >= 2.0) && (t_prev <= 2.0))
    character1->set_desired_action("walk");

if ((t >= 8.0) && (t_prev <= 8.0))
    character1->set_desired_action("walk_lo");
```

```

if ((t >= 14.0) && (t_prev <= 14.0))
    character1->set_desired_action("prone_aim");

if ((t >= 18.0) && (t_prev <= 18.0))
    character1->fire_weapon_n_times(4, 1.0f);

```

6. In your update loop, update the scenario with the current time, in seconds:

```
scenario->update(t);
```

7. In your draw loop, draw your DI-Guy characters. (In scene graph style renderers, this step is not necessary.)

```
scenario->draw();
```

8. When finished, clean up the character, the scenario, and DI-Guy:

```

scenario->destroy_character(character1);
diguy_destroy_scenario(scenario);
diguy_deinitialize();

```

1.1.3. Network with Other Simulators (DIS or HLA)

A common mode of operation for DI-Guy SDK is to visualize human characters simulated by a remote application. A typical case is when you connect your simulation to a DIS or HLA network. (DI-Guy support HLA 1.3, HLA 1516 (SISO DLC), and HLA Evolved.)

Another common case is where your simulator is a HOST-IG style simulator, often connected using a CIGI pipe. DI-Guy provides support for broadcasting, receiving, displaying, and controlling characters in networked simulations. In addition to functions that deal with standard DIS and HLA lifeform functions, DI-Guy provides custom PDUs for DIS and the DI-Guy FOMs for HLA, that extend these protocols to take full advantage of DI-Guy's capabilities. (DI-Guy has a custom FOM for HLA 1.3 and HLA 1516 and FOM modules for HLA Evolved.)

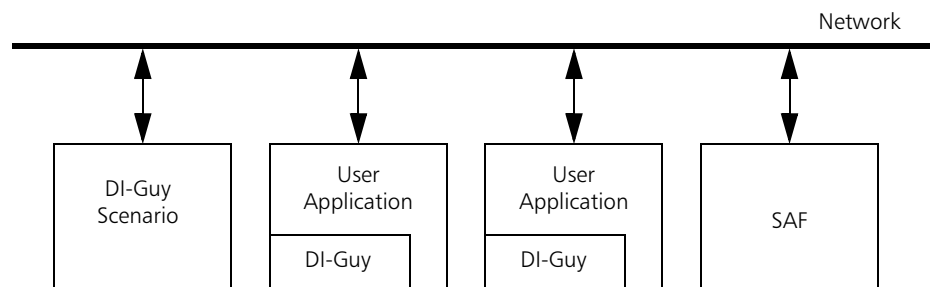


Figure 1-2. Using DI-Guy in a networked simulation

The following example shows how to network your DI-Guy characters to a DIS network. It is in *./programming_examples/diguy_networking/DIS*. This example uses VR-Link (sold separately) to provide the networking layer. DI-Guy is designed to interface easily with whatever networking layer you use, so if you want to hook up your own networking middleware to DI-Guy, you can do so.

1. Initialize DI-Guy:

```
diguy_ogl_initialize(NULL);
```

2. Initialize DIS:

```
diguy_module_net_initialize();  
diguy_net_module = get_net_interface();
```

3. Create a DI-Guy scenario:

```
diguyScenario *scenario = diguy_create_scenario();
```

4. Call networking functions to specify that we are receiving. Set the network host ID and go online:

```
diguy_net_module->set_publish_local_chars(0);  
diguy_net_module->set_host(2);  
diguy_net_module->go_online(scenario);
```

5. In your update loop, get updates from DIS and update the scenario with the current time in seconds:

```
diguy_net_module->update();  
scenario->update(t);
```

6. In your draw loop, draw the scenario. Scene graph style renderers do not require this step.

```
scenario->draw();
```

7. When you shut down, do it in the proper order.

```
diguy_net_module->go_offline();  
diguy_net_module->update(); // don't forget this update  
diguy_destroy_scenario(scenario);  
diguy_module_net_deinitialize();  
diguy_deinitialize();
```

The SDK also has the following examples of networking using HLA:

- ♦ For HLA 1.3: *./programming_examples/diguy_networking/HLA*.
- ♦ For HLA 1516: *./programming_examples/diguy_networking/HLA1516*.
- ♦ For HLA Evolved: *./programming_examples/diguy_networking/HLA1516e*.

1.2. Querying and Setting Actions in a Pipeline Setting

In a pipeline setting, (which may be a Host-IG configuration, or a DIS/HLA network), a DI-Guy user can be confronted with one or both of the following questions:

- ♦ I am running DI-Guy for constructive purposes on my host system. What is the best way to query DI-Guy to understand when an action has changed so that I can send it down the pipe?
- ♦ I am running DI-Guy for visualization purposes on my IG system. What is the best way to command new actions as they come down the pipe?

Table 1-1 shows the basic ways a DI-Guy character can have its action change on a host.

Table 1-1: Ways that actions can change

Character Mode	The way the character has its action changed	Visual Result
PATH	Character has action beads on the path.	Smooth, nice transitions.
PATH or FREE	<code>set_desired_action()</code>	Smooth, nice transitions, but action will start after current action loop is complete for maximum visual quality.
PATH or FREE	<code>force_action(<action name>, DIGUY_DEFAULT_FLOAT, 1, 0.5f, ...)</code>	Smooth transition that starts immediately. Quality of transition is not as nice as <code>set_desired_action()</code> but typically is quite good.
PATH or FREE	<code>force_action(<action name>, DIGUY_DEFAULT_FLOAT, 0, 0.0f, ...)</code>	Instant transition with discontinuity.

1.2.1. Host Side

The recommended way to handle the host side is to use the character callback function `diguyCharacter::CALLBACK_ID_CURRENT_ACTION_CHANGED()`. This function is called when a new action should be commanded down the pipe.

A less desirable approach is to create a string variable called `previous_action` for each character, and then at every time step:

1. Query the character's actions using `get_current_action()` and `get_desired_action()`.
2. If the `current_action` equals the `desired_action` AND the `current_action` does not equal the `previous_action`:
 - If the `current_action` does not equal the `desired_action`, send a new action packet with a smooth bit set to 1.
 - Otherwise, send a new action packet with a smooth bit set to 0.

1.2.2. IG Side

The recommended way to handle the IG side is to always call `force_action()` with the smooth bit determining whether to make the third and fourth arguments be (1, 0.5) or (0, 0).

Another way to do this is to use `diguyMotionEngineSnapshot::get_motion_engine_snapshot()`. This function returns a structure that tries to encode all the major variables that the motion engine is currently using to manage the primary action, and related transition values. The `multi_channel` example demonstrates this approach.



We also recommend the use of a DI-Guy Guide using the drift or adaptive algorithm for each IG character if the IG does not receive position and orientation information every timestep. The `guide_example`, in *`./programming_examples/opengl`*, shows how the guide can be used to best effect.



2. DI-Guy SDK Concepts

This chapter introduces concepts you need to understand to use the SDK.

How DI-Guy Works	2-2
The Character Viewer	2-2
DI-Guy Skeletal Structure	2-3
The Basic DI-Guy Function Calls	2-3
A Brief DI-Guy Class Overview	2-6
Best Practices for Using DI-Guy	2-7
Working with the diguyScenario and diguyCharacter Classes	2-9
The diguyScenario Class	2-9
The diguyCharacter Class	2-14
The DI-Guy Graphics API	2-23

2.1. How DI-Guy Works

DI-Guy is designed to simplify the task of adding lifelike human characters to real-time simulations. The goal is to allow you to work at a high level, concentrating on telling characters where to go and what to do, while the software and content handles the less interesting, but critical low-level details such as:

- ♦ Joint angle control and kinematics.
- ♦ Smooth and realistic motion generation derived from motion capture.
- ♦ Graphics hierarchy management.
- ♦ Load management.
- ♦ Realistic geometry and texture files.

2.1.1. The Character Viewer

DI-Guy ships with a license-free application called the DI-Guy Character Viewer. It lets you view all the characters and motions shipped with DI-Guy.

- To open the Character Viewer, on the Start menu, choose **All Programs → DI-Guy Applications 13 → DI-Guy Character Viewer**.

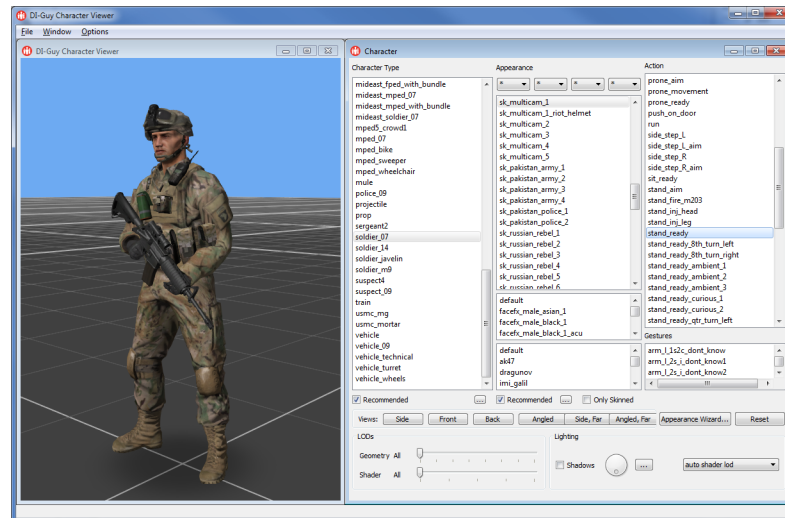


Figure 2-1. Character Viewer

2.1.2. DI-Guy Skeletal Structure

Figure 2-2 illustrates a representative skeletal structure for DI-Guy characters, including names of joints and links.

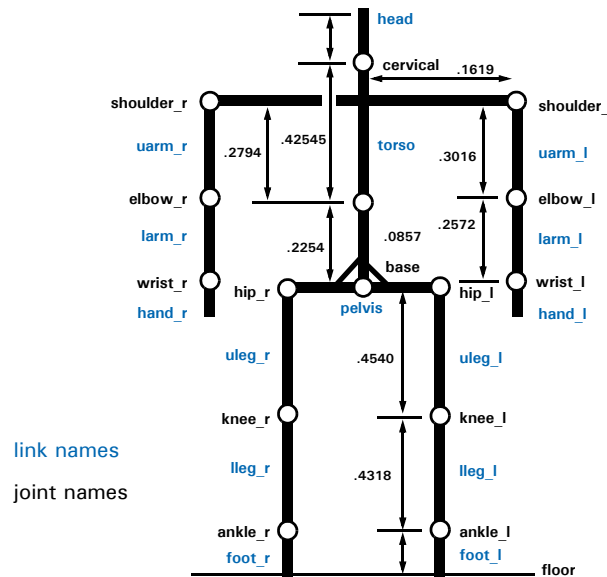


Figure 2-2. Typical DI-Guy skeletal structure

Not all DI-Guy characters share the same skeletal structure, so you should rely on the DI-Guy API query functions to determine topology, joint names, and link names whenever possible.

2.1.3. The Basic DI-Guy Function Calls

This section introduces function calls that are common to almost every DI-Guy implementation. Understanding these calls will also help you understand the basic concepts.

General Initialization and De-initialization Functions

Working with DI-Guy starts and ends with the initialization and de-initialization function calls. The initialize call is different for each rendering environment, as follows:

- ♦ OpenGL – `diguy_ogl_initialize()`.
- ♦ DI-Guy graphics API – `diguy_graphics_initialize()`.



If you are using Vega Prime, do not call the initialize/deinitialize calls directly, Vega Prime's plug-in architecture takes care of them automatically.

The de-initialize call is the same for all renderers – `diguy_deinitialize()`.

Module Initialization and De-initialization

In addition to the basic DI-Guy initialization and de-initialization function calls, there are other initialize and de-initialize calls that are sometimes needed, based upon whether or not you are using a particular DI-Guy module. These include:

- ♦ Lua scripting: `diguy_script_lua_initialize()`, `diguy_script_lua_deinitialize()`.
- ♦ DI-Guy AI: `diguy_ai_initialize()`, `diguy_ai_deinitialize()`.
- ♦ DIS and HLA networking: `diguy_module_net_initialize()`, `diguy_module_net_deinitialize()`.



Use of DIS or HLA is determined by the libraries you link against.

- ♦ Expressive Faces: `diguy_facefx_initialize()`, `diguy_facefx_deinitialize()`.
- ♦ OpenAL sound: `diguy_sound_openal_initialize()`, `diguy_sound_openal_deinitialize()`

Scenario Management Functions

The DI-Guy has functions to:

- ♦ Create scenarios.
- ♦ Populate the scenario.
- ♦ Update the scenario.
- ♦ Draw the scenario.

The C++ class *diguyScenario* is typically your top-level DI-Guy container class. There is actually a higher class, *diguyApp*, useful for application-level and multiple scenario control, but it is useful to think of the scenario as the top-level object for almost all your needs. You need to create a scenario when programming with DI-Guy. Use `diguy_create_scenario()` to do this.

Once the scenario is created, you need to populate it. You can populate a scenario in the following ways:

- ♦ By instantiating characters using `diguyScenario::create_character()`. Once a character exists, command its behavior using functions such as `diguyCharacter::set_action()`.
- ♦ By hooking up DIS or HLA networking to display characters published on the network. This code will use `diguyScenario::create_character()`.
- ♦ By loading a scenario file (*.dss*) created in DI-Guy Scenario using `diguyScenario::load()`.
- ♦ Indirectly using crowd creation functions in DI-Guy AI.

A call to the `update()` function advances time, performs any scheduled operations, runs logic, and updates the values for all the articulations of each character. Update is usually called every frame. To perform an update, call `diguyScenario::update()`.

DI-Guy works with both immediate mode and scene graphs. For scene graph renderers, the scene graph automatically calls the rendering functions for the characters. For immediate mode renderers, you must use a DI-Guy function call to perform the rendering. So for immediate mode rendering, be sure to include a call to the function `diguyScenario::draw()`. Similar to update above, you can also call `draw()` on your individual characters.

2.1.4. A Brief DI-Guy Class Overview

The two most important DI-Guy classes are:

- ♦ *diguyScenario*. This class provides access, either directly or indirectly, to all the other objects in DI-Guy. A scenario is the collection of all the characters used during the simulation, as well as information about controlling time and behavior.
- ♦ *diguyCharacter*. This class refers to a single DI-Guy entity, be it human, animal, vehicle, or prop. The *diguyCharacters* are usually the key points of interest in a scenario. A *diguyScenario* contains an array of all its *diguyCharacters*. A *diguyCharacter* can be referenced by index or by name from *diguyScenario*. A *diguyCharacter* has a rich interface for querying and specifying its behavior and appearance.

The following classes are also important:

- ♦ *diguyGraphicsLink*. A *diguyCharacter* is a hierarchy of links and joints. Think of the links as the bones of the characters while joints are the articulations. The links themselves do not have any geometry directly associated with them. Geometry is associated with shapes, which in turn are associated with links.
- ♦ *diguyGraphicsShape*. DI-Guy encapsulates its textured geometry in objects called shapes. One or more shapes are associated with each link, so when a link is “drawn”, it is actually its array of shapes that are drawn. For example, a head link might have two shapes associated with it: a human head and a helmet. Skinned DI-Guy appearances do not have traditional shapes, but instead have a mesh associated with the entire skeleton.
- ♦ *diguyCharacterPath*, *diguyPathShape*, *diguyWaypoint*

The most basic way of controlling a *diguyCharacter* is to specify its position and action. Both the position and the action can be in one of two modes: path or free. A character in path position and path action mode will have its position and actions governed by a *diguyPaths*. The shape of the path is determined by its path shape, which in turn is defined by waypoints. Waypoints are 3-D points through which the path shape must pass, as well as information about the angle and acceleration the path shape takes when it goes through the waypoint. A path shape can be owned by a scenario, and thus shared between multiple characters. Action beads on the path direct the characters on what actions to perform. A character in path position and free action mode will follow its path, but the user specifies the actions the character is performing. A character in free position and free action mode is typically given a starting position and then accumulates its position as it travels as you directly specify which action the character is performing. The combination free position/path action mode is not allowed.



DI-Guy AI adds a path planner and high level behaviors to improve movement on paths.

- ♦ *diguySceneObject*. A scene object is a static (that is, non-moving) object in the scenario to which characters can sometimes be ground clamped. Scene objects are typically sections of terrain, such as a small town.
- ♦ *diguyView*. A view is a graphics window in which the scenario and its characters are animated. Views have a current camera and may have settings for visual effects, such as lighting and fog. Providing view information can sometimes be useful for DI-Guy, and is required when using the Load Manager.
- ♦ *diguyApp*. This class is a higher object than *diguyScenario*, and is useful for managing multiple scenarios and other super scenario assets.

2.1.5. Best Practices for Using DI-Guy

The following sections describe best practices for using create functions and using path and action modes.

Instantiate DI-Guy Classes Using create() Functions

When using DI-Guy, the method for instantiating new classes is to call create() functions from a higher class. Do not construct and destroy DI-Guy classes using new and delete functions. These constructors are private and cannot be called. Often these classes will automatically get destroyed when the higher class is destroyed. For example, to create a character, you should use the create_character() function call from the *diguyScenario* class.

Use the Path and Action Mode that Works Best for Your Application

DI-Guy behaves differently based on the position mode and action mode in use. The position mode determines how a character's location is set and the action mode determines how the character knows when to perform actions. Both modes can be set to either free or path. [Table 2-1](#) shows how the two modes interact:

Table 2-1: Path and action modes

Mode	Interaction
path position + path action	Characters follow a <i>diguyCharacterPath</i> that also commands what actions they perform.
path position + free action	Characters follow a <i>diguyPathShape</i> , but have their actions commanded directly by the user through the API.
free position + free action	Characters move based on the actions commanded by the user through the API, accumulating position and orientation.
free position + path action	Illegal.

DI-Guy characters can also be controlled using methods described in *DI-Guy AI Users Guide*. Please consult that manual for information about creating and controlling autonomous characters that use artificial intelligence to determine their behavior and avoid obstacles and other entities.

2.2. Working with the *diguyScenario* and *diguyCharacter* Classes

The *diguyScenario* and *diguyCharacter* classes are the two most powerful classes in DI-Guy. Most of your development will center on working with these two classes. The following sections describe some of the key functions and concepts of each class.

2.2.1. The *diguyScenario* Class

The following sections describe important types of functions in the *diguyScenario* class.

Update and Draw Functions

The `update()` and `draw()` functions are key functions used in even the simplest DI-Guy application.

The `update()` function advances time forward to time `t` and new position and joint values are calculated for all the characters. No rendering happens in `update()`.

On immediate mode rendering systems, rendering is performed in the `draw()` function. The `draw` function calls two sub-functions, `draw_pass1()`, which draws the opaque objects in the scene, and `draw_pass2()`, which draws the transparent objects. You can call the subfunctions directly if you need that level of control.

Functions are provided for enabling and disabling culling of characters that are outside the view frustum. Characters are surrounded by bounding spheres.

Working with *.dss* Files

If you save a scenario created with DI-Guy Scenario, it creates a *.dss* file. You can use the `load()` function to load a *diguyScenario* from the *.dss* file.

Time Control and Playback Functions

The `reset()` function rewinds the scenario back to the start time (`t-in`) and resets a number of features such as signals. Reset is almost always the preferred way to restart a scenario, as opposed to just setting `t` to zero.

Use `get_t()` if you want to know what time the scenario thinks it is. Accessor functions such as `get/set_tout()`, `get/set_playback_loop()`, and `get_tick_dt()` allow users to set and get various values associated with time control of a DI-Guy scenario.

The concept of associating a clock time-of-day to `t-in` is available in DI-Guy and can be accessed through the `tin_time_of_day` family of functions.

Performance Functions

The following functions improve loading performance.

- `diguyScenario::preload_character_type()`
- `diguyScenario::preload_gesture()`
- `diguyScenario::preload_appearance()`.

These functions let you avoid problems at runtime that can occur if new character data is loaded when a character or gesture is called for the first time. These functions load the data during startup, when delay is less important.

You can also improve performance when updating DI-Guy. You can use a multi-threaded control API in *diguyApp*. For details, please see `diguyApp::set_num_threads()`. The `get/set_ticks_can_be_dropped()` and `get/set_max_ticks_behind_till_dropped()` functions let you tune scenarios that have trouble running in real time. For more information about improving performance, please see Chapter 4, [Maximizing DI-Guy Performance](#).

General Query Functions

The *diguyScenario* class has functions that let you get the following information about DI-Guy characters and their capabilities:

- Number of different character types available.
- The number of appearances or actions available for a given character type.
- The speed, distance, direction, or facing angle of a particular action.
- The number of gestures available.

When a number of items are available, there are functions for getting those items by index.

Character Functions

Perhaps the most important function of the *diguyScenario* class is to act as a container for DI-Guy characters.

- `get_num_characters()`. Reports how many characters are in the scenario. Functions are then available to retrieve these characters either by name or by index.
- `create_character()`. Creates a new character.
- `destroy_character()`. Destroys a character.
- `set_iguy_character()`. Designates a character as being driven by a joystick. We call a DI-Guy character that is controlled by a joystick an I-Guy.

Path Shape and Waypoint Functions

Path shapes are the splines that underlie paths in DI-Guy. The *diguyScenario* class has functions for creating, indexing, copying, and destroying path shapes. Once a path shape has been defined, DI-Guy characters can follow them. The path shapes are in turn composed of waypoints, which can also be manipulated through the *diguyScenario* interface.

Signal Functions

Signals are useful as a means to call event handlers. You can create and destroy signals, retrieve the number of signals, retrieve a signal using its index, or retrieve a signal using its name. All of the signals can be reset using `reset_signals()`.

Sound Functions

You can create and destroy sounds, retrieve the number of sounds, retrieve a sound using its index, or retrieve a sound using its name. All sounds can be reset using `stop_all_sounds()`.



Sounds are not played directly. They are used as templates for sound instances. Please see `diguyCharacter::play_sound()` and `diguyCharacter::create_sound_instance()` for more details.

Group Functions

Groups are subsets of the characters belonging to the scenario. They can be very useful as a way to organize characters. For example, when networking, combatants are classified as belonging to groups `net_friendly`, `net_opposing`, or `net_neutral`. You can create and destroy groups, retrieve the number of groups, retrieve a group using its index, or retrieve a group using its name.

Sensor Region Functions

Sensor regions are user-definable cubic regions that sense when characters enter and leave them, and also know how many characters are in them. You can retrieve the number of sensor regions and retrieve a sensor region using its index. Sensor regions can have callbacks associated with them for when characters enter or leave.

DI-Guy AI supports arbitrarily shaped regions.

Scene Object Functions

Scene objects are stationary objects (usually terrains) to which waypoints can be ground clamped.

You can create and destroy scene objects, retrieve the number of scene objects, retrieve a scene object using its index, or retrieve a scene object using its name. Scene objects can be enabled or disabled (that is, rendered).

View Functions

Views are windows in which rendering occurs. You can find a view by name, retrieve the primary view, or retrieve a secondary view by index.

Camera Functions

Each view has a camera associated with it that defines how the 3D scene is viewed in the window. Multiple camera settings can be contained in a scenario, and the camera can be set to the values in any of those camera settings. Functions are available to query the number of camera settings, or get a camera setting by index or name. The function `load_camera_settings()` sets the primary view's camera parameters from the specified camera setting.

There is also a set of functions for setting or querying the relationship of script events to the camera.

Fog and Light Functions

diguyScenario has functions for managing fog and light settings, including querying the number of settings, retrieving a setting by index or name, and manipulating the values of the settings. These functions are relevant only if you want to program while running inside of DI-Guy Scenario.

Information Popup Functions

Information popups are boxes with text that appear when running a scenario inside of DI-Guy Scenario. An API is available to manage the popups.

Variable Functions

You can declare user variables that are attached to the *diguyScenario* class. An API is available to manage the variables, including declaring, setting, naming, finding, and using the variables.

Facial Expression Functions

You can create, delete, and manage the facial expressions associated with Expressive Faces software.

Callback Functions

You can use the `add_callback()` function to have a function called when any of the following scenario-level events occur. Some of the events are specific to DI-Guy Scenario.

- ♦ `CALLBACK_ID_CREATE`
- ♦ `CALLBACK_ID_DESTROY`
- ♦ `CALLBACK_ID_RESET`
- ♦ `CALLBACK_ID_WAIT_CURSOR_SHOW`
- ♦ `CALLBACK_ID_WAIT_CURSOR_HIDE`
- ♦ `CALLBACK_ID_LOAD`
- ♦ `CALLBACK_ID_SAVE`
- ♦ `CALLBACK_ID_LOAD_SCENARIO_FILE`.

For additional callbacks see *diguyScenario.h*.

Default Character Callbacks, Altitude Functions, and other Default Callbacks

The function `add_default_character_callback()` lets you assign particular character level callbacks to all subsequent characters added to the scenario. Similarly, `set_default_character_altitude_function()` lets you assign a particular altitude function to any character subsequently added to the scenario. There are similar methods for assigning callbacks to paths, sensor regions, signals, views, cameras, fog, and lights.

Event Handler Functions

DI-Guy event handlers are similar to callbacks, but have a name with which they can be referenced. `map_event_handler_to_callback_id()` maps an event handler function to a particular callback ID. The API provides functions for registering various event handlers tied to signals, sensor regions, character events, and scenario events.

History and Review Functions

You can save and load history for after action review of your scenario. Call `set_history_type()` to `DIGUY_HISTORY_TYPE_COMPLETE`, since this is a DI-Guy SDK application. When playing back scenarios, other calls made to DI-Guy API functions may be ignored, unless you are at a point in the scenario for which no data had been recorded.

LOD Functions

DI-Guy supports levels-of-detail for graphics (where characters are drawn with various numbers of polygons), motion (where characters can limit which joints are updated), and shaders. Activation, ranges, and other methods of parameterizing the LODs are available in the API.

Meta–Action Functions

The basic concept of meta-actions is that you should be able to query a character for his “closest action” without knowing exactly what actions or their names are available by describing the direction-of-travel, speed, posture, and variant of the action he would like to use. For example, you might ask for a moving forward motion that goes four MPH in an upright posture. For a soldier, the “jog” action might be returned, while for the female_pedestrian, perhaps “powerwalk” would be returned. Variant can mean different things, for example, “funky” versus “stiff” might be a way of differentiating between two different types of walking.

2.2.2. The *diguyCharacter* Class

The following sections describe important types of functions in the *diguyCharacter* class.

General Accessor Functions

- ♦ `get/set_name()`. Gets or sets the name of the character.
- ♦ `get/set_type_name()`. Gets or sets the character type of a character.
- ♦ `get_scenario()`. Gets the scenario the character belongs to.
- ♦ `get/set_enabled()`. Gets or sets a character’s enabled status.
- ♦ `get/set_drawn_by_scenario_flag()`. Determines whether the character gets drawn when its parent scenario gets drawn.
- ♦ `get/set_current_tin()`, `get/set_current_tout()`. Gets or sets the time range during which the character will be active.
- ♦ `get/set_position()` and `get/set_orientation()`. Gets or sets geographic position or orientation of the character.
- ♦ `get/set_position_double()`. Gets or sets position when the character needs to be far away from the origin.
- ♦ `set_position_relative_to_parent()` and `set_orientation_relative_to_parent()`. Used when parenting the character to another character.
- ♦ `set_initial_position()` and `set_initial_orientation()`. Sets the position of a character when a reset occurs. This is for free position mode only. In path position mode, the start of the path is where the character will go on a reset.
- ♦ `get/set_scale()`. Gets or sets the scale of the character along the three major axes.
- ♦ `is_group_member()`. Returns whether the character belongs to a particular group.

Update and Draw Functions

- ♦ `update()` advances time forward to time `t` and new position and joint values are calculated for the characters. No rendering happens in `update()`.
- ♦ `get/set_t_controlled_by_scenario_t()` checks to see if your character has its time controlled by the scenario (this is the default).
- ♦ Rendering is performed in the `draw()` function on immediate mode rendering machines. The draw function calls two sub-functions, `draw_pass1()`, which draws the opaque objects in the scene, and `draw_pass2()`, which draws the transparent objects. You can call the subfunctions directly if you need that level of control.

Appearance Functions

- ♦ `get/set_appearance()` and `get/set_current_appearance()` let you get and set a character's appearance. Current appearance (as returned by `get_current_appearance()`) may be different than the base appearance. This base appearance is the starting appearance of the character before any calls to `set_current_appearance()` have been made. This means that if the scenario is reset, the appearance will revert to the base appearance.

i

DI-Guy currently prohibits changing the appearance of a character to a new appearance associated with a different actor once a scenario is underway. This is due to DI-Guy methods for reducing foot slippage. It is therefore important to use `set_appearance()` before advancing time. Choose an appearance associated with the actor with whom all anticipated future appearances are also associated.

- ♦ `get/set_current_head_appearance()` allows you to swap heads on the character.
- ♦ `get_current_hand_item()`, `set_hand_item()`. These functions let you get and set the character's hand item.

Dying

- ♦ `die_now()`. Tells a character to die as soon as possible. Characters execute their dead action. All aiming, gazing, pointing, head nodding, gestures, and sounds stop.
- ♦ `get_dead()`. Gets the dead state of a character.
- ♦ `revive_now()`. Resuscitates a character.

Performance Functions

Use of shader LODs can improve performance. Use the `get/set_graphics_lod()` and `get/set_shader_lod()` functions.

Characters can be culled in or out of the scene. You can set or view a character's bounding radius with `get/set_bounding_radius()` and `get/set_default_bounding_radius()`. DI-Guy Character Viewer can also display them. Please see the culling examples for how to use these functions inside your application.

The "Up" Vector

Normally "Z" is up for a character, but occasionally you may want to alter the up vector. A good example would be a vehicle like an airplane banking as it made a turn. Use the `get/set_up_vector()` to access this value.

Action and Speed Functions

- `set_speed()`. Sets a goal velocity after calling `set_desired_action()`. You can undo the speed setting by calling `set_speed()` with `DIGUY_DEFAULT_FLOAT` or by subsequently calling `set_desired_action()` or `force_action()`.
- `get_speed()` retrieves the current speed of the character.
- `set_t_scale_factor()`. Dilates time around the character. Use `unset_t_scale_factor()` to turn it off.
- `get/set_desired_action()`. Puts the character into free action mode. A character that was in path action mode will now ignore subsequent action beads on his path. Characters that are in path position mode should use the `retain_path_shape` argument to determine whether they will leave path position mode for free position mode.
- `force_action()`. Forces a new action to be executed immediately. By contrast, `set_desired_action()` allows the current action to complete its cycle before transitioning.
- `force_action_duration()`. Used when the character is in path position mode. It forces the character to perform an action for a certain period and then revert to the path.
- `force_action_and_path_shape()`. Forces the action and puts the character on the designated path shape.
- `get_time_to_transition()`. Returns the time till a desired action begins.
- `get_action_mode()`. Returns the action mode of the character.

The following function support animation blend tree functionality:

- ♦ `get/set_animation_velocity()` and `get/set_animation_local_velocity()`. Sets a generic parameter in world/local space that can be mapped to a blend tree driver. The locomotion actions are driven by these values allowing one action to go many different directions.
- ♦ `get/set_animation_angular_velocity()`. Sets a generic parameter that can be mapped to a blend tree driver to control rate of turning in locomotion actions

For more information about animation blend trees, please see [“Animation Blend Trees,”](#) on page 3-3. For more information about these functions, see the Animation Blend Tree Parameters section in *diguyCharacter.h*.

Parenting Functions

`set/unset_parent()` allows you to parent a character to another character. Characters move relative to the parent character, not global space.

Link Functions

- ♦ `get_link_position()`. Gets the location and orientation of a particular link of a character. `get_link_position_double()` returns a double precision answer when the character is far from the origin. The `get_link_position_with_offset()` and `get_link_position_with_offset_double()` functions allow you to get an offset into that link.
- ♦ `get_link_relative_position()`. Gets the position and orientation of the link relative to the parent link. There is also a `with_offset` version.
- ♦ `get_base_link()`, `find_link()`, and `get_link_at_index()`. These functions get particular links.

LOD Functions

- ♦ Call `get/set_lod_ranges()` to access the graphics LOD ranges of the character. You can use `set_graphics_lod()` to manually set the graphics LOD of the character. You should turn off automatic LOD switching at the scenario level when you do this. Use the DI-Guy Character Viewer to visually and textually review the LODs of the character. DI-Guy characters have up to 7 LODs, with individual meshes having up to 4 LODs, typically decimated according to the 50-30-30 rule (LOD 2 has 50% of the polygons of LOD 1, and so on).
- ♦ `get/set_motion_lod()` Gets or sets the motion LOD state of the character. Coordinate use of the set function with the Load Manager.

User Data

`get/set_user_data()`. Lets you attach your own classes or structures to a DI-Guy character and access them.

Path Functions

- ♦ `set_distance_along_path()`. Positions a character at a particular distance along a path when in path position mode. `leave_path()` makes a character change from path position mode to free position mode.
- ♦ `get_position_mode()`. Tells you whether the character is in free or path position mode.
- ♦ *diguyCharacter* has functions for determining how many paths are associated with the character and you can retrieve them by name or index.
 - `push_path()`. Appends a path to a list of paths the character will follow.
 - `force_path()`. Takes a character off the current path and puts it on the input path.
 - `force_partial_path()`. Designates a start position on the input path.
- ♦ `resume_interrupted_path()`. Allows you to get back to a path after a `set_desired_action()` or `force_action()` removed you from the path and the `retain_path_shape` argument was set for those functions.
- ♦ `create_and_force_bridge_path()`. Creates a temporary path from your character's current position on a path and switches the character onto it. This temporary bridge path then transitions the character onto the new path.
- ♦ `create_path()` and `create_simple_path()`. Allow runtime generation of a path. The former creates an empty path, the latter a two waypoint path.
- ♦ `jump_to_action_bead()`. Causes the character to jump forward along its current path to the named action bead.

Sound Functions

Call `play_3d_sound()` to have your character play a sound that “follows” your character. `play_sound()` plays an “ambient” (positionless) sound.

Sound Instance Functions

A sound instance is, roughly speaking, a playing copy of a sound. A single sound can be layered on top of itself many times. Each of these layers is a sound instance. For example, a single gunshot sound might be used for every soldier's gun. Since these guns might be fired nearly simultaneously, each gun, when it fires, has its own instance of the gunshot sound. Any time a sound is played, a sound instance is being used. Generally, these sound instances are managed internally to DI-Guy and you need not worry about them if you use only `play_sound()` or `play_3d_sound()`. But if you need finer control over sound, you will have to work with sound instances.

Sound instances are generally most useful for longer sounds, such as the engine sound of a vehicle. With such sounds, you may want to play them for a shorter time than their natural duration. You need to be able to stop them before they play out naturally. You also might want to repeat a relatively short sound so that it will fill up a desired amount of time. You can do both of these things with sound instances.

Please see the SDK reference for details on using sound instances.

Perception Functions

Use `is_within_distance_n_of_character()` to query whether any other characters are near your character. `is_within_distance_n_of_member_of_group()` queries whether any characters that belong to a particular group are near your character.

Other functions for detecting characters include:

- ♦ `get_nearest_character()`
- ♦ `is_character_visible()`
- ♦ `get_nearest_active_character()`
- ♦ `get_nearest_active_character_in_group()`
- ♦ `check_character_visibility()`
- ♦ `check_point_visibility()`.

For details, see *diguyCharacter.h*.

Callback Functions

Use the `add_callback()` function to have a function called in response to character-level events. Please refer to the SDK Reference Manual for a list of callbacks. Types of callbacks include upon create, upon destroy, and pre and post weapon fire.

Event Handler Functions

`map_event_handler_to_callback_id()` maps an event handler function to a particular callback ID. DI-Guy event handlers are similar to callbacks, but have a name with which they can be referenced. The API has functions for registering various event handlers tied to signals, sensor regions, character events, and scenario events.

Aim, Gaze, Pointing, and Nodding Functions

Characters can aim, gaze, or point by being directly commanded or by reaching a bead on the path they are traveling.

Aim, gaze, and point have many functions for specifying how to aim, gaze, and point, whether it is at a point in space, a local direction, or at another character or link. Functions are also available for determining the status or duration of the aim, gaze, or point, and to terminate these actions. Query functions allow you to determine a character's particular ability to aim, gaze, and point.

A smaller API is available for nodding and head shaking for default, non-animated heads. The Expressive Faces module offers dynamic facial animation.

Altitude Functions

While a character is in path position mode, its altitude (position in Z direction) is most often determined by the path. Characters in free position mode often need their altitude to be determined by another method. This can be done by setting the position manually (using the `set_position()` call, or semi-automatically by registering an altitude function. An altitude function is a callback function that gets called once per character update. Based on the character's updated X and Y position, the callback can calculate and return an appropriate Z position. This is commonly done using some type of graphics intersection testing.

Weapon Functions

DI-Guy characters that have weapons can fire them. The SDK has functions for determining the muzzle flash and sound to play when firing. Use `set_weapon_fires_live_rounds()` to determine whether the character might hit or kill other characters. The kill zone has a near, far, and weapon spread that is programmable.

Characters can fire a weapon using the `fire_weapon()` or `fire_weapon_n_times()` function.

By default, when a character gets shot, it dies. However, you can use the `CALLBACK_ID_IMPACT` and `CALLBACK_ID_CROWD_MEMBER_IMPACT` callbacks to override this behavior and create custom damage models. These callbacks are called when a character has been hit. `diguyCharacter::get_last_impact_record()` contains a pointer to the impact information. If a character has this callback the standard behavior (killing the character) is skipped and the system assumes the end user has handled the impact.

The *diguyImpact* class contains detailed information for the result of a weapon fire. Consult this class for more information for supporting advanced weapon effects.

Gesture Functions

A gesture is a motion a character can do that temporarily overrides a subset of the character's joints. Gestures are designed to supplement the stock set of character motions. Most gestures are available to all characters.



Figure 2-3. Gestures

DI-Guy Scenario has an interactive mode that allows gestures to be easily previewed, right click on any character and do show Gesture Previewer. For more information about gestures, please see [“Gestures,”](#) on page 3-6.

DIS Functions

DI-Guy characters support setting and querying information about how they are broadcast or received from a DIS networking session. Attributes such as whether a character should be published, its lifeform state, weapon state, network marking, and entity number can all be set or queried.

Working with Guides

A guide is a mechanism for controlling the position, orientation, or action of a character that is in free position mode and free action mode. Guides are most often used by networking software, where you only get intermittent information about a character's position and actions, or formations. In either case, there is usually a current state and a goal state, and the guide is used to try to minimize the difference in the two in a believable manner. Most guides look at the desired position, velocity, and orientation of the character and do what is necessary to get the character there (or at least closer). How a guide accomplishes this depends on:

- ♦ The guide algorithm being used.
- ♦ The parameters set for that algorithm.
- ♦ The difference between current and desired settings.

The desired position and orientation for a character can be set by `set_desired_position()` and `set_desired_orientation()`. They may also be implicitly set if that character is in a formation. Guides can be explicitly added to a character by calling `add_guide()` or `create_guide()`. They can also be implicitly added to a character if that character is called into a formation. Guide parameters are set by calls to `diguyCharacter-Guide::set_guide_algorithm_float_parameter()`.

The `set_desired_velocity()` function lets you do dead-reckoning.

Once a guide has been added to a character it can affect the position, orientation, and/or action of a character until the guide has been removed. Guides can be explicitly removed from a character by calling `remove_guide()` or `remove_all_guides()`. They can be implicitly removed by the break-up of a formation, or by an automatic removal guide acquiring the target destination.

Currently the following algorithms are offered as guides:

- ♦ Exact
- ♦ Follow1
- ♦ Follow2
- ♦ Drift1
- ♦ Adaptive (*this is typically the best choice for your guide).

Please see the SDK reference for more information on particular guide algorithms and functions to activate and manage guides.

Variable Functions

Just like scenarios, characters can have user variables associated with them. A set of functions is available for querying, creating, setting, and using user variables. Variables can be integers, strings, or floats.

2.3. The DI-Guy Graphics API

The DI-Guy Graphics API is a class-based system for integrating DI-Guy graphics with rendering systems. It ships with examples for OpenGL, OpenSceneGraph, and DirectX 9. Many DI-Guy customers use the DI-Guy Graphics API to customize DI-Guy to their specific renderer. The DI-Guy Graphics API lets users access the building blocks of DI-Guy character graphics such as shaders, polygons, textures, links, and so on.

The DI-Guy Graphics API is used to:

- ♦ Support proprietary rendering environments.
- ♦ Integrate DI-Guy tightly with image generators.
- ♦ Render DI-Guy characters with special lighting and rendering effects.
- ♦ Implement advanced sensor views, such as IR and radar views.
- ♦ Perform collision detection on DI-Guy characters.

The DI-Guy Graphics API provides a variety of functions to support these needs and to allow DI-Guy to be used in almost any rendering environment. The DI-Guy Graphics API supports both scene graph renderers (for example, – OpenSceneGraph, Vega Prime) and immediate mode renderers (for example, OpenGL).

DI-Guy Graphics API programming should be done by experienced C++ programmers comfortable with object-oriented design and familiar with their particular renderer. If you fit this description, you will find the DI-Guy Graphics API a well-architected programming solution for customizing DI-Guy graphics to your renderer.

The DI-Guy Graphics SDK Reference contains a detailed description of the objects, classes, and functions that make up the Graphics API. Implementers should use this document, along with the open source examples for OpenSceneGraph, OpenGL, and DirectX9 delivered with DI-Guy, to guide their development of a DI-Guy Graphics API solution tailored to their specific rendering situation.



3. The Motion Pipeline

This chapter describes how DI-Guy processes motions.

The Motion Pipeline.....	3-2
The Initial Stage.....	3-2
Gazing and Pointing.....	3-4
Aiming.....	3-5
User Pose Overrides.....	3-5
Gestures	3-6
Head Nodding and Shaking	3-6

3.1. The Motion Pipeline

When DI-Guy determines the joint animation (also called degrees-of-freedom) of a character during an update, it follows the sequence describes in [Table 3-1](#).

Table 3-1: Stages of motion pipeline

Stage	Description
Initial	Retrieve initial pose based on action and aiming state.
Stage 0	Reserved for users.
Stage 1,2	Gazing, Pointing (except eyes).
Stage 3	User Pose Overrides.
Stage 4	Gestures.
Stage 5	Nodding, shaking of head.
Stage 6	Motion texture.
Stage 7	Gaze (eyes).
Stage 8	Unused.
Stage 9	Reserved for users.

It is useful to understand this progression, so that you can understand how the motions are constructed. You may even want to insert your own modifications to the final pose of the character in Stage 0 or Stage 9.

Inverse kinematics can take place after any of these stages.

3.1.1. The Initial Stage

When DI-Guy advances in time, it checks the underlying motion files to see if the animation is looping, or repeating itself, or if it is transitioning to another animation. If it is transitioning, it blends the animation into the new motion.

The gaze system can trigger a change in orientation if the orientation parameter is set to true. When a character needs to look at something beyond his normal gazing range, the character shuffles its feet to rotate its body so that it can achieve the desired gaze.

Aiming is performed by interpolating in joint space between two or four aim poses that define the extremes of the aiming space. These values are also calculated during the initial stage.

Animation Blend Trees

DI-Guy has an animation blend tree system that can allow actions to map to a data driven graph of underlying motion files. This graph can treat a collection of motion capture data as procedural inputs into a mathematical system that allows us to synthesize new motions on the fly.

The following nodes are available:

- ♦ A 2 motion blender that can play an animation on a subset of bones.
- ♦ 1D and 2D Blender nodes that can play multiple animations at once and interpolate their characteristics.
- ♦ Selectors nodes that can switch between a list of child animations based on a key/value pair.

These tools are useful to allow DI-Guy to create a ‘walk aiming’ motion by blending the lower body of walk with the aim upper body, but still logically treat them as one action. Or to allow the smooth interpolation between a character that is walking and one that is running, or walking, running, and strafing.

3.1.2. Gazing and Pointing

Use the Gaze and Pointing API to have a character stare or at point at a particular location in space ([Figure 3-1](#)).



Figure 3-1. Gazing

Gazing and pointing can be commanded in a variety of ways from the API. A character can gaze and point:

- ♦ At a point referenced locally.
- ♦ At a point referenced globally.
- ♦ At a character.
- ♦ At a link on a character.
- ♦ Using an azimuth and elevation.

API functions allow you to access and modify how quickly the gaze or pointing takes effect and how interpolation is performed.

Gazing can be as minimal as eyes rotating towards a target, or as dramatic as the character stepping to turn around, while altering the hips, back, and cervical joints. Pointing affects the left shoulder and left elbow. (Use gestures for right arm pointing.)

Please see the DI-Guy SDK Reference for a function-by-function description of the gazing and pointing features available in DI-Guy.

3.1.3. Aiming

Use the Aim API to have a character equipped with a weapon aim that weapon. Like gazing and pointing, aiming can be specified in a variety of ways. A character can aim:

- ♦ At a point referenced locally.
- ♦ At a point referenced globally.
- ♦ At a character.
- ♦ At a link on a character.
- ♦ Using an azimuth and elevation.

The Aim API lets you read and set how quickly an aim takes effect, as well as how interpolation is performed.

Characters need to be playing an aiming action to enable the aiming API. Thus, a character performing a “stand” action cannot aim, but a character doing a “stand_aim” can.

Please see the DI-Guy SDK Reference for a function-by-function description of the aiming features available in DI-Guy.

3.1.4. User Pose Overrides

You can directly set or modify a part of a pose or the entire pose of a character using a pose override. Pose override essentially views the state of the character as an array of values for each degree-of-freedom. You capture the current state and modify it. An array of weights is available that indicates for each degree-of-freedom how much the pose override value should affect the value already calculated in the motion pipeline up to the point in time that the pose override is applied.

Please see the DI-Guy SDK Reference for a detailed functional description of user pose override capabilities.

3.1.5. Gestures

Gestures are motions that are applied to a subset of the degrees-of-freedom of a character. You can order the character to perform gestures. Gestures augment any motion already being performed. Layering motions in this way can be useful for creating signals and are critical when wanting to create a believable talking character. Most gestures are available to all characters.

Gestures have stages, which are played in order. Before stage 2 of the gesture is played, stage 1 must be played. Sometimes a stage is repeated multiple times before the gesture moves on to the next stage. Stages are numbered 1 to 3, not 0 to 2.

An example of a three stage gesture could be a wave. In stage 1, the gesture raises the character's arm. In stage 2, the arm waves back and forth. Stage 2 may be repeated multiple times to get the character to wave more than once. Stage 3 lowers the arm back to the character's side.

Some gestures use the concept of channels. If a gesture has more than one channel, usually the gesture is represented in a different manner in each channel, perhaps to indicate different emotive states. An example of a two channel gesture could be a chop beat gesture for emphasizing speech. Channel A may have a small, tentative chop gesture, while Channel B contains a strong, emphatic chop gesture. If the weights of channels A and B are 1 and 0, respectively, the final gesture would be identical to the small chop. If the weights are 0 and 1, respectively, the final gesture would be the emphatic chop. If the weights are 0.5 and 0.5, respectively, the final gesture would be a combination – partly small, partly emphatic. As the gesture progresses, the weights can be varied to add even more flavor and nuance to the resultant gesture.

Please see the DI-Guy SDK Reference for an abstract and functional description of these capabilities.

3.1.6. Head Nodding and Shaking

DI-Guy characters have head nodding and head shaking functions. These are simple functions that are appropriate if you do not need the flexibility that comes with manipulating the head using gestures.

Please see the DI-Guy SDK Reference for a functional description of these capabilities.



4. Maximizing DI-Guy Performance

This chapter explains performance features of the DI-Guy SDK.

Techniques for Maximizing DI-Guy Performance	4-2
 Using the Load Manager	4-4
 Using DI-Guy as Part of a Networked Image Generator	4-5
 Notes on Compressed Geometry, Textures, and Preloading	4-6

4.1. Techniques for Maximizing DI-Guy Performance

DI-Guy provides tools, techniques, and functions for improving runtime performance. You can use these tools to create simulations that look great and run quickly. In this section performance refers to the efficient update and drawing of DI-Guy characters so as to maximize graphics frame rate (draw) and update rate, and minimize CPU load.

The following basic techniques help scale performance of DI-Guy characters:

- ♦ Draw Culling. If a character is not visible in the view frustum, eliminating a draw call to that character will improve performance.
- ♦ Update Culling. If a character is not visible in the view frustum, eliminating an update call to that character will improve performance. Non-visible characters still need to be updated periodically so that their general position is known (and thus whether they have entered the view frustum). It is often also important to stagger these updates to maintain performance consistency.
- ♦ Shader LODs. DI-Guy supports use of shader LODs to improve graphic performance. This include bump mapping, per pixel shaders, per vertex, and unlit shaders. Shader LODs let you balance visual quality against performance.
- ♦ Instancing. The instancing system lets you add 100s of identical characters to scene without compromising performance.
- ♦ Graphics LOD. Graphics LOD specifies how detailed a model to display. In [Figure 4-1](#), we see 4 LODs, with polygon counts of 6500, 3250, 1280, and 425 respectively. LOD4 is clearly an unacceptable appearance if the character is in the foreground, but if the character is far away, it is visually indistinguishable from LOD 1, but takes only a fraction of the time to render.

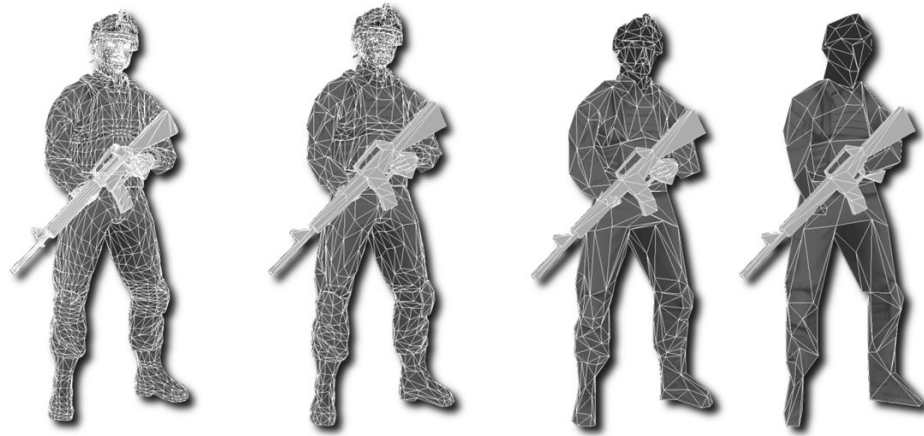


Figure 4-1. Models at different levels of detail

You can set the graphics LOD of a character using the *diguyCharacter* function call `set_graphics_lod()`. However, typically LODs are controlled automatically. You can query and set the distances that define LOD ranges on a per-character basis using the `diguyCharacter::set_lod_ranges(...)` and `get_lod_ranges(...)` function calls.

The DI-Guy Character Viewer is an excellent way to review a model's LODs, both visually and numerically.

- ♦ Multi-Threading. DI-Guy takes advantage of multiple cores to animate its characters. Use `diguyApp::set_num_threads()` to control how many different threads DI-Guy uses for animating.
- ♦ DI-Guy Load Manager. The DI-Guy Load Manager allows you to apply the culling and LOD methods described previously to a single view application in a simplified manner so that you do not need to handle the details. The Load Manager is described in “[Using the Load Manager](#),” on page 4-4.
- ♦ Shader LODs. Shaders can be instrumented for characters farther away from the origin.
- ♦ Advanced Performance Techniques. Advanced performance techniques can increase your DI-Guy performance. The use of skinned versus segmented characters often affects the relevance of these techniques. A benchmark program is included to help measure the effect of these advanced performance options. Advanced techniques include:
 - Disabling motion blends.
 - Disabling update.
 - Disabling draw.
 - Disabling motion accumulation.
 - Enabling cached pointers.
 - Enabling quaternion precomputation.
 - Enabling axis motions optimization (aim)

4.1.1. Using the Load Manager

When there are many characters in a scene, the DI-Guy Load Manager (class *diguy-LoadManager*) uses update culling and motion LODs to improve runtime performance. Characters far from the camera are made to articulate less often than characters close to the camera. Through this process the Load Manager reduces the total amount of CPU processing used to update DI-Guy characters, with little to no degradation of the visuals.

The Load Manager divides the world into four zones: near zone, intermediate zone, far zone, and out-of-view (Figure 4-2). Characters in the near zone articulate fully at full rate. Characters in the medium and far zones articulate at a reduced rate and use motion LODs to limit the joints articulated. Characters in the out-of-bounds zone are not articulated at all, and have their location calculated at a reduced rate. (The Load Manager does not support multiple viewports.)

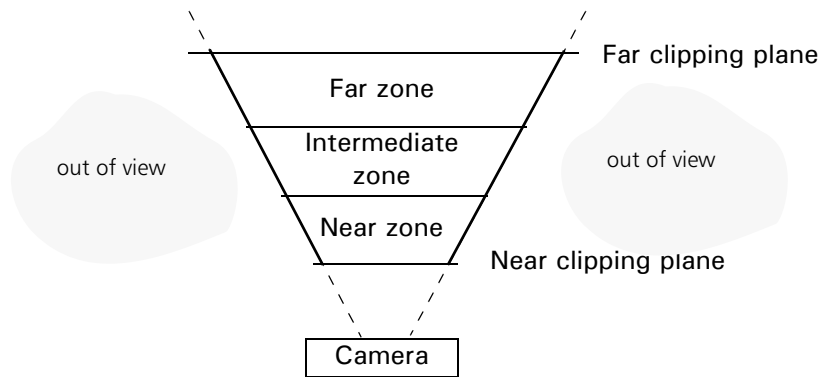


Figure 4-2. LOD zones

Enabling Load Management

By default, the Load Manager is disabled. To enable the Load Manager, create it using the *diguyApp* function `create_load_manager(...)`. This function returns a pointer to an instantiated *diguyLoadManager* class.

You must also associate a camera with the Load Manager so that it knows which characters are visible and which are not. Do this by calling the Load Manager's member function `add_view_camera()`. Any time the properties of the view change, the Load Manager must be advised using the function `change_view_camera_settings()` or `move_camera()` or using the callback `camera_changed()`.

Setting Zone Boundaries

Call the *diguyLoadManager* member function `set_zone_bounds()` to set the distances from the camera that define your zones. The default distances are 10 and 25 meters respectively.

Setting Zone Parameters

Call the *diguyLoadManager* member function `set_zone_parameters()` to assign the percentage of time the character's location is updated, percentage of time the joints are updated, and the motion level of detail (MLOD) for any zone. The following are the default settings expressed in terms of calls to the Load Manager functions:

```
Set_zone_bounds(10, 25);
Set_zone_parameters(0, 100, 100, 1); # Near Zone: 100% pos, 100% joint,
MLOD 1
Set_zone_parameters(1, 100, 50, 2); # Intermediate Zone: 100% pos, 50%
joint, MLOD 2
Set_zone_parameters(2, 100, 20, 5); # Far Zone: 100% pos, 20% joint, MLOD
5
```

Setting the Maximum Update Period for Out-of-Frustum Characters

Characters located outside of the view frustum will by default have their location updated at a lower rate (once per second) with no articulation when using the load manager. Use the `set_max_update_period()` function to specify a longer or shorter interval between position updates.

4.1.2. Using DI-Guy as Part of a Networked Image Generator

A typical DI-Guy use case is to use the DI-Guy SDK to provide the lifeform visuals for an image generator that is part of a distributed simulation receiving entity state information using DIS or HLA. You can improve performance by using the following techniques:

- Draw culling, update culling, motion LODS, and load management. Draw culling, update culling, and motion LODs are powerful methods for improving performance. The DI-Guy Load Manager is an easy way to apply those methods with minimum coding. But the DI-Guy Load Manager is not the correct solution, in some cases.

If your application provides multiple views of a scene, do not use the DI-Guy Load Manager. You will have to create your own load manager to manage the DI-Guy draw culling, update culling, and motion LODs functions.

Do not use any load management if all the characters are close by and always in the view frustum.

- Graphic LODs. Set your LOD ranges in each view carefully to maximize draw time performance by reducing polygon count and draw calls.
- Disable motion accumulation. Do this if you are setting the position of your DI-Guy character every frame. If you are only periodically updating, use the drift guide and do not disable motion accumulation.
- Shader LODs. Picking the simplest shader that you need can greatly increase the speed of drawing. Please see the *diguyGraphicsShaderTechnique* class for more details.

4.1.3. Notes on Compressed Geometry, Textures, and Preloading

Compression of geometry and textures has been emphasized in DI-Guy to improve load-time performance, disk footprint, memory footprint, and runtime performance.

DI-Guy ships with compressed geometry (*.bdg*) files. DI-Guy Data CD3 includes the uncompressed version of these models. By default, when loading a model that does not have compressed geometry, DI-Guy will try to compress and save the *.bdg* file for future use.

DI-Guy supports multi-texturing. To minimize texture memory, it automatically compresses *.png* and *.rgb* textures to *.dds* files.

Normal map textures use the DXT5_nm texture compression format. DI-Guy reference shaders only support compressed normal map textures.

DI-Guy does not load characters, motions, geometry, and textures when it reads the configuration file during initialization, it just catalogs them to know what is available. This saves time during loading, but at a performance cost during runtime. To ensure that no frames are dropped when a character is first used at runtime, there are various preloading methods available that force loading at startup.

The following functions load geometry and texture for an appearance:

- `diguy_*_initialize()` loads appearances with “preload=true”.



By default DI-Guy always sets characters to preload=false.

- `diguyScenario::preload_appearance()` loads the specified appearance.
- `diguyScenario::preload_character_type()` loads the default appearance.
- `diguyScenario::create_character()` loads the default appearance and the specified appearance.
- `diguyCharacter::set_current_appearance()` loads the specified appearance.

The following functions load motion data for character actions:

- `diguyScenario::preload_character_type()`
- `diguyScenario::create_character()`.

The following functions load motion data for a gesture:

- `diguyScenario::preload_gesture()`
- `diguyCharacter::create_gesture()`
- `diguyCharacter::execute_gesture()`
- `diguyCharacter::execute_1stage_gesture()`
- `diguyCharacter::execute_2stage_gesture()`
- `diguyCharacter::execute_3stage_gesture()`.

DI-Guy keeps references to things it has loaded, and does not reread files.



5. Networking with DI-Guy

This chapter describes use of DI-Guy with DIS and HLA.

Networking with DI-Guy	5-2
Extending DIS and HLA for High Fidelity Visual and Motion Correlation .	
5-2	
DIS Networking Using the SDK.....	5-3
Controlling the Update Rate	5-4
How DI-Guy Custom PDUs are Defined.....	5-4
DI-Guy Entity Mapping in the Absence of Custom PDUs.....	5-6
HLA Networking.....	5-7
RTI Compliance	5-7
Setting up DI-Guy to use HLA	5-8
Enabling HLA Networking in DI-Guy.....	5-10
Controlling the Update Rate	5-11
DI-Guy Entity Mapping	5-11
RPR FOM Elements Used by DI-Guy	5-12
DI-Guy FOM Extensions for HLA	5-12
Troubleshooting Your HLA Application	5-13
Fire and Detonate Interactions in DI-Guy	5-13
Network Configuration Parameters for DI-Guy Scenario.....	5-15
Database Coordinates and DI-Guy Networking.....	5-18
Setting the Exercise Location in DI-Guy	5-19
UTM Database Coordination Translation	5-19
Controlling DI-Guy Scenario Remotely Using Custom PDUs.....	5-19

5.1. Networking with DI-Guy

DI-Guy supports DIS and HLA. It has functions that map the standard entity, fire, and detonation events from DIS PDUs and HLA RPR FOM messages into DI-Guy function calls. DI-Guy also has functions that map behavior created using the DI-Guy API into calls that produce standard DIS or HLA traffic.

5.1.1. Extending DIS and HLA for High Fidelity Visual and Motion Correlation

DI-Guy extends DIS and HLA to enhance the functionality and versatility of human characters in distributed simulations. These extensions are called DI-Guy Custom PDUs for DIS and the DI-Guy FOM Extension for HLA. The custom PDUs and DI-Guy FOM let you transmit the exact appearance and character type of a character, as well as the exact action being performed. Even user-generated custom appearances and motions are supported.

The DI-Guy SDK does not include DIS and HLA code to send and receive DIS or HLA traffic. Applications that use DI-Guy typically have their own networking layer that is either created by these customers in-house or provided by third party products such as VR-Link. DI-Guy has networking functions that can be hooked up to any networking layer in addition to special function calls designed specifically to be used with VR-Link. (The DI-Guy Lifeform Server and DI-Guy Scenario with the networking option provide full networking capabilities. Internally they use VR-Link.)

Applications such as VR-Forces, Presagis STAGE, and Vega Prime use DI-Guy. When they receive DI-Guy data over the network they can use the full repertoire of DI-Guy characters and animations to display high-quality characters. Other applications that interact with DI-Guy-enabled applications over the network will receive DI-Guy data, but their ability to interpret and visualize that data may vary.

The custom PDUs and DI-Guy FOM add information to the standard DIS and HLA RPR FOM messages. If you are running DI-Guy applications with non-DI-Guy applications, applications using DI-Guy will take advantage of the additional information. Those without DI-Guy will process the standard information in the messages and ignore the DI-Guy content.

If you are using a DI-Guy Lifeform Server (the DI-Guy Scenario application with networking and AI), custom PDUs can be used to remotely control the Lifeform Server.

5.2. DIS Networking Using the SDK

DI-Guy broadcasts and responds to standard DIS entity state PDUs. It can also broadcast and respond to DI-Guy Custom DIS PDUs. DI-Guy also broadcasts and responds to standard DIS fire and detonation PDUs. Examine the programming example *diguy_networking/DIS*, detailed below:

1. Before the DI-Guy scenario object is constructed, call:

```
diguy_module_net_initialize();
```

and retrieve the net interface:

```
diguyNetInterface * diguy_net_module = get_net_interface();
```

2. To join the exercise once the DI-Guy scenario object is constructed, call:

```
diguy_net_module->go_online()
```

3. Once per frame or tick of the simulation clock during your simulation, call:

```
diguy_net_module->update()
```

4. When you want to leave the DIS exercise, call:

```
diguy_net_module->offline()
```

5. At least once after calling `diguy_net_module->go_offline()` and before calling `diguy_module_net_deinitialize()`, call:

```
diguy_net_module->update()
```

This allows the offline command to be processed, so that objects to be destroyed in Step 6 are unused.

6. After the DI-Guy scenario object has been destroyed, call:

```
diguy_module_net_deinitialize()
```

Please see *diguy_module_dis.h* for more information about these functions, as well as a listing of other DIS-related functions offered by DI-Guy.



The *diguyNetInterface* class acts as a wrapper around either a DIS or an HLA network interface. The type of interface that is created depends on which library is linked against.

The *./programming_examples/diguy_networking/DIS/net_send* example demonstrates broadcasting DIS. The *./programming_examples/diguy_networking/DIS/net_recv* example demonstrates receiving DIS.

5.2.1. Controlling the Update Rate

DI-Guy periodically broadcasts DIS entity state PDUs for each character. It broadcasts PDUs when the dead-reckoned error in position of an entity exceeds one of the following:

- ♦ The translation_threshold (default 0.05m).
- ♦ The rotation_threshold (default 0.2 radians).

PDUs are also sent if the timeout threshold is reached (default 5 seconds). The timeout rate can be altered by calling `diguyNetInterface::set_timeout_interval()`.

These threshold parameters are specified in the configuration file `./custom/config/diguy/scenario/network.cfg`.

5.2.2. How DI-Guy Custom PDUs are Defined

The IEEE standard 1278.1 defines DIS PDUs. The first record of a PDU is the PDU Header Record, defined as follows:

```
typedef struct PDUHeader_s {
    U8      m_version ; // 4 for DIS 2.0.4,
    U8      m_exerciseID ;
    U8      m_PDU_kind ;
    U8      m_family ;/* unused in DIS Version 3 */
    U32     m_time ;
    U16     m_length ;
    U16     m_unused ;
} PduHeader_t;
```

Setting the `m_pdu_kind` variable to a value between 220 and 255 designates a custom PDU. You can change the `m_pdu_kind` by editing the `cust_pdu_kind = 220` line in `network.cfg`. The default value is 220. All your DI-Guy custom PDUs should use a single common number. `m_version` should be:

- ♦ 4 – For DIS 2.0.4.
- ♦ 5 – For IEEE 1278.1.
- ♦ 6 – For DIS 2.1.4 or IEEE 1278.1a.

The overall DI-Guy Custom PDU is defined by the structure:

```
struct diguyCustomPdu {
    PDUHeader_t  m_header;           /* Header, defined above */
    U16          m_bdimessageversion; /* Depends on MessageID */
    U16          m_bdimessageid ;    /* Message ID */
    EntityID_t   m_entityID ;        /* Entity ID defined below */
    U8           m_datasize ;        /* Size of m_data */
    U8           m_data[] ;          /* Message data */
};
```

where:

- ♦ `m_pdu_header`. The PDU record header described previously.
- ♦ `m_bdimessageversion`. Depends on the message ID. Is used to make messages backwards compatible. Currently `m_bdimessageversion` is 7 for all message IDs except `SETCHARACTION`, which must be 8. If you ever receive a DI-Guy PDU with an unsupported message version, ignore the message.
- ♦ `m_bdimessageid`. A zero-based enumeration, with the following mnemonics defined:

```
enum {  
    DIGUY_CUSTOM_PDU_ID_ESCAPE_RESERVED = 0,  
    DIGUY_CUSTOM_PDU_ID_SET_CHAR_TYPE,  
    DIGUY_CUSTOM_PDU_ID_SET_CHAR_APPEARANCE,  
    DIGUY_CUSTOM_PDU_ID_SET_CHAR_ACTION,  
    DIGUY_CUSTOM_PDU_ID_SET_CHAR_VISIBLE,  
    .  
    .  
};
```

i

A second set of message IDs, the DI-Guy Scenario Control PDUs, are discussed in “[Controlling DI-Guy Scenario Remotely Using Custom PDUs](#),” on page 5-19. The message IDs listed here are for Live Reckoning.

- ♦ `m_entityID`. Identifies the site, host, and entity, with three 16-bit values.
- ♦ `m_datasize`. Describes how many bytes are in `m_data`. If `m_data` is longer than 255 bytes, this value is zero.
- ♦ `m_data`. Void data whose format depends on the value of `m_bdimessageid` and `m_bdimessageformat`. `m_data` can be any combination of short integers, long integers, 32-bit floating point values, and null-terminated strings. For instance, if the format is long, short, string, string, then the first 4 bytes are a long integer, the next 2 bytes are a short integer, followed by a null-terminated string, and then another null-terminated string.

The most important messages are for the type, appearance, and action. For a complete list, see the `diguyCustomPDUType` enumeration in *diguy_dis_constants.h*.

For further information and type definitions, consult IEEE 1278.1.

i

In DIS PDUs, the network representation of multi-byte integers is big-endian. Therefore, on Intel architectures, multi-byte integers sent to and received from the network must have their byte order reversed.

5.2.3. DI-Guy Entity Mapping in the Absence of Custom PDUs

DI-Guy using custom DIS PDUs supports the full range of characters available in DI-Guy by specifying the character type and appearance within the PDU. In the absence of custom PDUs, a mapping configuration file specifies:

- The character type and appearance DI-Guy uses to display entities being received.
- The DIS entity type and parameters to broadcast for locally simulated characters.

If neither custom PDUs nor an entity map specify the character and appearance, DI-Guy displays and sends the entity as either:

- An American soldier (lifeform).
- An American M1A1 Abrams tank (vehicle).

Specify the desired mappings in `./custom/config/diguy/scenario/network_model_map.cfg`. The default configuration file is `./config/diguy/scenario/network_model_map.cfg`.

DI-Guy offers a network model map application as part of DI-Guy Scenario that has an easy-to-use GUI to update this file.

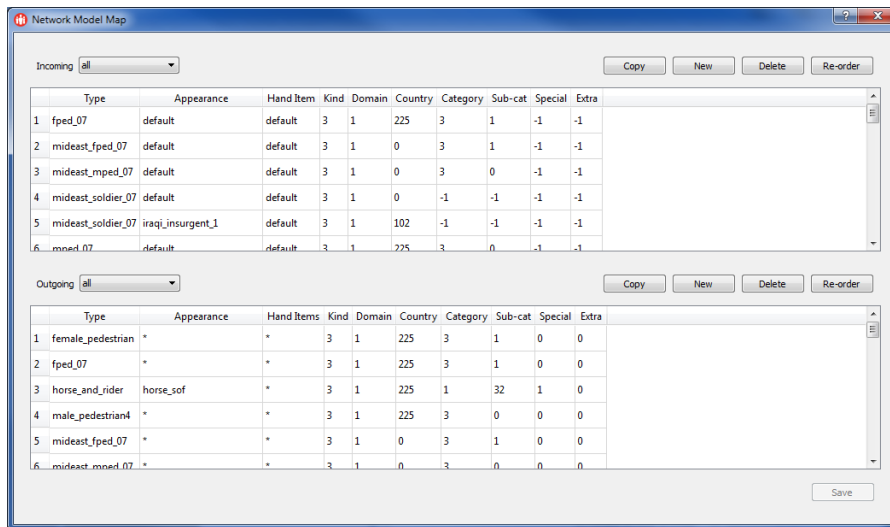


Figure 5-1. Model map application

Each entry line in `network_model_map.cfg` corresponds to a DIS Entity Type Record of a DIS Entity State PDU.

- A DIS Entity Type Record contains 7 fields that describe the type of entity to which the PDU refers.
- The order of lines in the mapping file is not important.

In the mapping file:

- ♦ -1 is a wildcard meaning any number.
- ♦ Asterisk (*) is a wildcard meaning any string.
- ♦ Wildcards cannot precede non-wildcards in any line of the mapping file.
- ♦ All lines are considered during mapping, and the most specific match (the one with the fewest wildcards) is used.
- ♦ A line with all wildcards to the left of the second equals sign denotes a default mapping, used when no more specific mapping exists.
- ♦ If there is no match in the table, DI-Guy uses an American soldier for lifeforms and an American M1A1 Abrams tank for platforms.
- ♦ Comments on the end of each line are ignored.
- ♦ The two word pairs (for example, “vehicle2 helicopter1” or “soldier2 default”) refer to the DI-Guy character and appearance.

The following are some example entries in *network_model_map.cfg*:

```
entry = 1 1 225 2 1 3 0 = vehicle_turret m2a3_bradley_des_turret
entry = 1 1 225 2 1 3 1 = vehicle_turret m2a3_bradley_grn_turret
entry = 1 1 225 2 1 3 2 = vehicle_turret m2a3_bradley_des_turret
entry = 1 1 225 2 1 3 3 = vehicle_turret m1a2_abrams_des_turret
entry = 1 1 225 2 1 3 4 = vehicle_turret m1a2_abrams_grn_turret
entry = 1 1 225 2 1 3 5 = vehicle_turret t72_tank_des_turret
entry = 1 1 225 2 1 3 6 = vehicle_turret t72_tank_grn_turret
entry = 1 1 225 2 1 1 1 = vehicle_technical gaz_truck_w_ZSU23_turret
entry = 1 1 225 2 1 1 2 = vehicle_technical humvee_50cal_des_turret
entry = 1 1 225 2 1 1 3 = vehicle_technical humvee_50cal_grn_turret
```

5.3. HLA Networking

DI-Guy supports HLA (High Level Architecture) interoperation for HLA1.3, HLA 1516, and HLA Evolved. DI-Guy subscribes to and publishes standard RPR FOM (Real-time Platform Reference Federation Object Model) attributes, fire and detonation interactions, and DI-Guy Custom attributes described in this section.

The RPR FOM, augmented with the DI-Guy Custom attributes (FOM extensions), defines the DI-Guy-Enhanced FOM. The DI-Guy-Enhanced FOM allows more realistic simulation of humans than the RPR FOM.

5.3.1. RTI Compliance

DI-Guy does not include an RTI. To use HLA, you must obtain an RTI, such as the MÄK RTI, from an RTI vendor. To install an RTI, please follow the instructions provided by your RTI supplier. Usually your RTI will come with some simple test software. Make sure you get your RTI working independently before using the DI-Guy FOM.

5.3.2. Setting up DI-Guy to use HLA

Perform the following steps to configure DI-Guy to use HLA networking:

1. Verify that the RTI DLLs are in your path.

Once the RTI is installed, along with any other third party networking software, check to see if the installation has altered your \$PATH environment variable. Your \$PATH environment variable should contain the directory where the RTI's DLLs reside, specifically, *libFedTime.dll* and *libRTI-NG.dll*.

For the MÄK RTI, the name resembles *C:\MAK\makRti4.2\lib*.



Avoid having multiple different RTIs in your path as this will cause confusion.

2. Install and enable third party networking. The DI-Guy SDK does not have a low level networking layer. Your application must directly interface with the RTI or use middleware, such as VR-Link, to do so. DI-Guy includes networking modules (*net_send* and *net_rcv*) built using VR-Link that demonstrate how to connect to the network. They are in the *./programming_examples/diguy_networking* directories.
3. Choose a RID file.

Most RTIs require a RID (RTI Initialization Data) file. Use a RID file appropriate to your RTI. The RID file contains parameters for the configuration of the RTI. In practice, the name of the RID file need not end with *.rid*. For example, *rid.mtl* is the default name of the RID file for the MÄK RTI.

The RTI finds its RID file by looking at the environment variable *RTI_RID_FILE*. The value of the variable is the full path to the RID file, including the file name.

If DI-Guy finds that *RTI_RID_FILE* is not defined, it will ask you to define the environment variable. *./config/diguy* has some sample RID files.

4. Choose a FED file or FDD file.

In HLA 1.3, a FED file (Federation Execution Data) is a list of object attributes that will be transmitted over the HLA network for a given federation execution. DI-Guy requires a FED file that is RPR-FOM compatible. In HLA 1516 and HLA Evolved, the equivalent file is an FDD file.

DI-Guy recommends a FED file that includes the DI-Guy FOM Extensions.

The DI-Guy FOM Extensions are provided in *./config/diguy/DI-Guy.fed* for HLA 1.3; *./config/diguy/DI-Guy.xml* for HLA1516 and HLA Evolved.

The RTI finds its FED file by looking in the directory specified by the environment variable RTI_CONFIG. Set RTI_CONFIG to indicate the FED file's directory, but not the file name.

If DI-Guy finds that RTI_CONFIG is not defined, it assumes a value of *./config/diguy/networking*.

The FED file name used by the RTI is specified by the *federation_name* line in *./config/diguy/scenario/network.cfg*.

By default, the value on this line is DI-Guy. This specification causes the federation execution to be named DI-Guy.

When DI-Guy is not running, you may edit this line to change the name of the federation execution and the FED file name.

5.3.3. Enabling HLA Networking in DI-Guy

Once you have set up your computer for HLA networking as described in “[Setting up DI-Guy to use HLA](#),” on page 5-8, start networking by performing the following steps.

1. Before the scenario object is constructed, call:

```
diguy_module_net_deinitialize() diguyNetInterface * diguy_net_module =  
    get_net_interface();
```

2. To join the exercise once it is constructed, call:

```
diguy_net_module->go_online()
```

3. During your simulation, periodically call:

```
diguy_net_module->update()
```

4. When you want to leave the exercise, call:

```
diguy_net_module->go_offline()
```

5. At least once after calling `diguy_net_module->go_offline()` and before calling `diguy_module_net_deinitialize()`, call:

```
diguy_net_module->update()
```

This allows the offline command to be executed, thus freeing up objects so that they can be safely destroyed in step 6.

6. After the scenario object has been destroyed, call:

```
diguy_module_net_deinitialize()
```



The *diguyNetInterface* class acts as a wrapper around either a DIS or a HLA (HLA 1.3/HLA 1516) network interface, the type of interface that is created depends on which library is linked against.

Please see *diguy_module_hla.h* for more information about these functions, as well as a listing of other HLA-related functions.

If DI-Guy fails to join the exercise, it is usually because the RTI has failed to find the FED file or the RID file. Be sure that you have named your FED file based on the federation name, and that you have placed it in directory specified by RTI_CONFIG.

5.3.4. Controlling the Update Rate

Under HLA, the RTI largely determines when DI-Guy updates entity attributes. DI-Guy forces the maximum update interval of entities according to the following line in *networking.cfg*:

```
hla_force_update = 5
```

where:

- ♦ 5 – Indicates that attribute updates for all objects are sent at least every 5 seconds.
- ♦ 0 – Indicates DI-Guy will not force any updates, leaving the update rate entirely up to the RTI.

Please consult your RTI documentation for more details.

5.3.5. DI-Guy Entity Mapping

It is recommended that applications use the custom DI-Guy FOM extension for realistic human simulation.

If DI-Guy is broadcasting or receiving attributes from applications that do not use the DI-Guy custom FOM, DI-Guy uses the same entity mapping scheme as described in [“DI-Guy Entity Mapping in the Absence of Custom PDUs,”](#) on page 5-6.

5.3.6. RPR FOM Elements Used by DI-Guy

DI-Guy publishes and subscribes to the following attributes and parameters of the RPR FOM:

- ♦ BaseEntity.AccelerationVector
- ♦ BaseEntity.AngularVelocityVector
- ♦ BaseEntity.DeadReckoningAlgorithm
- ♦ BaseEntity.EntityType
- ♦ BaseEntity.EntityIdentifier
- ♦ BaseEntity.Orientation
- ♦ BaseEntity.WorldLocation
- ♦ BaseEntity.VelocityVector
- ♦ PhysicalEntity.DamageState
- ♦ PhysicalEntity.ForceIdentifier
- ♦ PhysicalEntity.Marking.
- ♦ LifeForm.PrimaryWeaponState
- ♦ MunitionDetonation.DetonationLocation
- ♦ CreateEntity.OriginatingEntity
- ♦ CreateEntity.ReceivingEntity
- ♦ CreateEntity.RequestIdentifier
- ♦ DeleteObjectInstance.ObjectInstance
- ♦ DeleteObjectInstance.Tag
- ♦ DeleteObjectInstance.FederationTime
- ♦ LocalDeleteObjectInstance.ObjectInstance.

5.3.7. DI-Guy FOM Extensions for HLA

DI-Guy includes two custom FOMs, one for HLA 1.3 and HLA 1516, and one for HLA Evolved. The DI-Guy FOM extensions define DI-Guy custom attributes that allow DI-Guy to subscribe to and publish data that enables high fidelity networked human simulation. This information describes character appearance, actions, and other useful information.

For details about the FOMs, you can view them in a text editor or XML editor. They are in *./config/diguy/networking*.

5.3.8. Troubleshooting Your HLA Application

If you are having problems running an HLA application with DI-Guy, try the following:

- ♦ Add the RTI's *bin* directory to your path.
- ♦ You can define RTI_CONFIG to be the directory containing your FED file. Otherwise, we assume a default.
- ♦ Define RTI_RID_FILE to be your RID file. Under the MÄK RTI, this may not be necessary.
- ♦ The hla_advisories setting in *network.cfg* must agree with your RTI configuration (your RID file).
- ♦ The federation_name setting in *network.cfg* must be the base name of your FED file. DI-Guy is the default.
- ♦ If you are running HLA with advisories, run *RTIExec.exe* to start the federation execution.
- ♦ If you are using VR-Link or the MÄK RTI, be sure that license management is working. You may need to define MAKLMGRD_LICENSE_FILE to point to your MÄK license server.

Example Environment Variables Batch File

Here is an example batch file for setting critical environment variables:

```
PTHREAD_DIR=$DIGUY/3rdparty/Cygnus/pthreads-w32-2-8-0
RTI_RID_FILE=C:/MAK/makRti4.04/rid.mtl
RTI_CONFIG=$DIGUY/config/diguy/networking
MAK_VRLDIR=C:/MAK/vrlink4.0.3
MAKLMGRD_LICENSE_FILE=@accordion
```

5.4. Fire and Detonate Interactions in DI-Guy

DI-Guy characters interact with fire and detonation PDUs when interoperating in DIS or HLA environments.

Entities simulated locally by DI-Guy can:

- ♦ Fire at other entities, regardless of whether those entities are simulated locally or remotely.
- ♦ Be fired upon and killed by other entities, whether simulated locally or remotely.

A DIS host or HLA federate that initiates a fire interaction models the trajectory and the impact or detonation location of the fired munition. Entities in the vicinity of a detonation interaction do damage assessment and respond accordingly.

- ♦ DI-Guy applications should initiate fire interactions whenever a locally simulated entity fires a supported weapon.
- ♦ DI-Guy applications should initiate detonation interactions when a munition fired by a locally simulated entity hits a target or detonates.
- ♦ DI-Guy applications should display muzzle flash and play a sound when they receive a fire interaction for a remotely simulated entity.
- ♦ DI-Guy applications should instrument entities simulated locally to respond to detonation PDUs.

DI-Guy Scenario does all of this out-of-the-box.

DI-Guy for Vega Prime performs items 1 and 3 in the previous list. You must implement items 2 and 4.

DI-Guy Scenario issues detonation interactions as follows:

- ♦ A local entity dies when intersected by a round.
- ♦ Local entities within a configurable (in *network.cfg*) detonation range of a grenade or bomb detonation die.
- ♦ An impact callback is generated.

DI-Guy Scenario displays muzzle flash and issues a fire interaction and a detonation interaction during the first pass of a looped simulation for local characters. If history is being saved, during playback you will only see a muzzle flash and no fire or detonation interaction are issued. If history is not being saved, during playback DI-Guy Scenario continues to issue fire and detonation interactions. (DI-Guy Scenario assumes that bullets travel instantaneously and along a straight line.)

DI-Guy for Vega Prime displays muzzle flash and issues a fire during the first pass of a looped simulation for local characters. If history is being saved, during the 2-N loops you will only see a muzzle flash and no fire or detonation interaction are issued. If history is not being saved, characters do not move and no fire or detonation interactions are issued.



Use the *diguyImpact* class to enhance fire and detonations in DI-Guy applications.

5.5. Network Configuration Parameters for DI-Guy Scenario

Table 5-1 summarizes the network configuration parameters in `./custom/config/diguy/scenario/network.cfg`. These parameters can be tuned to balance performance and fidelity. Be sure to save a copy of `network.cfg` if you modify it so that you can return to default settings if you encounter a problem.

Table 5-1: Network configuration parameters

Name and default value	Description
<code>velocity_smoothing_period = 0</code>	Time duration, in seconds, for character current velocity to match dead-reckoned velocity.
<code>position_smoothing_period = 0.1</code>	Time duration, in seconds, for character current position to match dead or live-reckoned position.
<code>translation_threshold = 0.05</code>	Permitted translation error, in meters, before DIS update is sent.
<code>timeout_interval = 12</code>	Time, in seconds, without packet traffic before an entity is assumed to be no longer on the network and is removed.
<code>iguy_velocity_smoothing_period = 0.3</code>	For characters not on a path, this time constant is used for smoothing their velocity before transmitting it.
<code>detonation_range = 5</code>	Range of lethality (kill radius) for incoming grenade or bomb detonation interactions not identified to the DI-Guy munition system. By default, this value is overridden by a more sophisticated detonation models using the DI-Guy munitions system, which essentially look up the type of munition and set values like detonation range programmatically.

Table 5-1: Network configuration parameters

Name and default value	Description
dr_algorithm = 2	Identifies the dead-reckoning algorithm to be used for translation and rotation error calculation: ♦ 0 = Other ♦ 1 = Static (Entity does not move.) ♦ 2 = DRM(F, P, W) ♦ 3 = DRM(R, P, W) ♦ 4 = DRM(R, V, W) ♦ 5 = DRM(F, V, W) ♦ 6 = DRM(F, P, B) ♦ 7 = DRM(R, P, B) ♦ 8 = DRM(R, V, B) ♦ 9 = DRM(F, V, B) F = Fixed, R = Rotational Frame P = Positional, V = Velocital Frame W = World, B = Body Coordinates
per_tick_publish_limit = 25	Limit number of entities to publish during a single tick. Helps keep performance uniform. Next tick will start publishing from entity list where previous publish stopped.
DIS Only	
dis_entity_custom_pdus = 1	Indicates whether DI-Guy custom PDUs are to be transmitted.
rotation_threshold = 0.2	Permitted rotation error, in radians, before DIS update is sent.
ip_address = 239.255.0.1	Address to which outgoing DIS traffic is sent. The default value indicates site multicast. In some situations, a value of 0.0.0.0 (broadcast) may work better, at the cost of affecting every machine on the network.
Port = 3000	Network port used for DIS networking.
exercise = 1	Exercise number.
site = 0	Site number.
host = 0	Host number.
sync_mode_enabled = 0	Indicates whether multi-screen network synchronization is enabled. In this mode, characters are neither broadcast nor received, but several machines running DI-Guy Scenario can be synchronized in their playback of scenarios.

Table 5-1: Network configuration parameters

Name and default value	Description
sync_master = 0	Sync mode only. Indicates whether this machine will be the sync master. If nonzero, this machine will send custom PDUs to keep remote instances of DI-Guy Scenario in sync.
slaved_camera = 1	Sync mode only. Indicates whether the camera on this machine will be slaved to the camera on the master machine. If not, the camera will be free, while playback itself will be synchronized with playback on the master machine.
roll_for_sync_mode = 0	Sync mode only. With a slaved camera, this value (in degrees) indicates how the local camera is oriented with respect to the remote camera on the master machine. This value indicates the roll difference between the two cameras. On the sync master, this value visibly alters the camera position, but it does not change the nominal position that the slaves see the master camera to be in. Generally, to maintain screen contiguity, this roll value should be set the same on the master and on all the slaves.
yaw_for_sync_mode = 0	Sync mode only. With a slaved camera, this value (a percentage) indicates how the local camera is oriented with respect to the remote camera on the master machine. This value indicates the yaw difference between the two cameras. For example, a value of 100 means yaw a full field of view to the left. A value of -50 means yaw half a field of view to the right. On the sync master, this value visibly alters the camera position, but it does not change the nominal position that the slaves see the master camera to be in. So, for example, to achieve two-screen horizontal contiguity, you could set the master's yaw to be 50, and the slave's yaw to be negative 50, or vice versa.

Table 5-1: Network configuration parameters

Name and default value	Description
pitch_for_sync_mode = 0	Sync mode only. With a slaved camera, this value (a percentage) indicates how the local camera is oriented with respect to the remote camera on the master machine. This value indicates the yaw difference between the two cameras. For example, a value of 100 means pitch a full field of view down. A value of -50 means pitch half a field of view up. On the sync master, this value visibly alters the camera position, but it does not change the nominal position that the slaves see the master camera to be in. So, for example, to achieve two-screen vertical contiguity, you could set the master's pitch to be 50 and the slave's pitch to be negative 50, or vice versa.
cust_pdu_kind = 220	A value from 220 to 255, used as the PDU kind ID for DI-Guy custom PDUs.
HLA Only	
hla_diguy_fom = 1	Indicates whether DI-Guy custom entity attributes are to be transmitted and received.
hla_force_update = 5	Maximum number of seconds between the transmission of HLA entity state updates. If zero, HLA entity state updates are not transmitted unless the state changes.
hla_advisories = 0	Indicate whether to activate advisories. This value must agree with your RTI configuration file.
federation_name = DI-Guy	HLA only. Name of the federation execution, as well as the base name of the FED file.

5.6. Database Coordinates and DI-Guy Networking

DI-Guy supports geocentric, UTM, and flat earth coordinate conversions. DI-Guy works in database coordinates. When DI-Guy communicates with a simulation using DIS or HLA, the locations of entities are specified to the network in geodetic coordinates.

With DI-Guy, you will not need to deal directly with geodetic coordinates, but you need to know how to situate your database on the surface of the Earth.

By default, the origin of your database is placed at zero degrees longitude, zero degrees latitude and at sea-level, according to the WGS84 ellipsoid representation of the earth.

5.6.1. Setting the Exercise Location in DI-Guy

Use these function calls to situate the database from the API when called before going online:

```
diguyNetInterface::set_lat_lon()  
diguyNetInterface::set_position_offset()
```

Please see the `net_send` and `net_rcv` programming examples for more information.

5.6.2. UTM Database Coordination Translation

For network transmission, database coordinates are translated into geodetic coordinates. This translation is not linear. Your database, which is taken as flat and rectangular, is curved to match the earth's surface. The origin of the database is mapped to some location within a UTM zone. From this point:

- ♦ Database X values correspond to relative easting.
- ♦ Database Y values correspond to relative northing.
- ♦ Database Z values correspond to relative altitude.

5.7. Controlling DI-Guy Scenario Remotely Using Custom PDUs

DI-Guy has custom PDUs that allow remote control of DI-Guy Scenario. Once DI-Guy Scenario has joined the network, it responds to custom PDUs to load scenarios, merge other scenarios into the current scenario, set time-of-day information, perform playback control functions (start, stop, reset, and so on), trigger scripts in the scenario, or even receive scripts composed by the sender. There are also some PDUs sent by DI-Guy Scenario that allow the sender to know when activities were completed.

The network master could also be another DI-Guy Scenario application if desired.

Please see `./include/diguy_dis_constants.h` for a list of custom PDUs, descriptions, and how to use them. This information is also available in the DI-Guy Reference.



6. Writing Plug-ins for DI-Guy Scenario

This chapter is a brief introduction to writing plug-ins for DI-Guy Scenario.

Writing Plug-Ins	6-2
Supported Plug-in Functions	6-2

6.1. Writing Plug-Ins

You can create plug-in DLLs that can work with DI-Guy Scenario at runtime. These can be useful for creating interfaces to custom hardware or adding intelligence to scenarios. Plug-ins require a separate license. Plug-ins should be compiled using the same compiler as DI-Guy Scenario. There is a detailed programming example in *./programming_examples/diguy_plugin*.

6.1.1. Supported Plug-in Functions

The following list of functions can be exported from a DLL plug-in and are recognized by DI-Guy Scenario. These functions are prototypes. You must implement the ones you want to use. However, you do not have to implement all the functions, just the ones you need. DI-Guy Scenario will automatically call these functions at the appropriate time. Plug-in functions should be declared with `__declspec(dllexport)` to make sure that the symbols are properly exported from the DLL. Please see the plug-in example in the DI-Guy OpenGL programming examples folder for an example of how to build a plug-in that will work with DI-Guy Scenario.

- ♦ Initialization functions:
 - `void initialize_plugin(diguyApp *app)`. Called once when the plug-in is being loaded. The call will happen while DI-Guy Scenario is starting up if the plug-in was specified on the command line, or during the `load_plugin()` call if the plug-in is being explicitly loaded by function call.
 - `void deinitialize_plugin(diguyApp *app)`. Called once, when the plug-in is being unloaded. The call will happen while DI-Guy Scenario is shutting down, or during the `unload_plugin()` call if the plug-in is being explicitly unloaded by function call.
 - `void initialize(diguyScenario *s)`. This function is called when the active scenario first becomes aware of the plug-in. This is when the scenario is created if the plug-in has already been loaded, or during the `load_plugin()` call if the plug-in is loaded after the scenario was created.
 - `void deinitialize(diguyScenario *s)`. Called when the scenario is deleted, or when `unload_plugin()` is called.
- ♦ Pre and post draw functions:
 - `void pre_draw(diguyScenario *s)`. This code is called once per frame before objects are drawn. Only code related to drawing should be here. Use `update()` for any other code you want to be called frequently.
 - `void post_draw(diguyScenario *s)`. This code is called once per frame after all other objects have been drawn. It is a good place to draw overlays or transparent objects.

i

Both draw functions work because they are called in a thread with a valid OpenGL context. Any OpenGL call should work, and querying and modifying the current OpenGL state should also work. It is your responsibility to make sure the OpenGL state is restored to what it was at the beginning of the function.

- ♦ Pre and post update functions:
 - void pre_update(diguyScenario *s). Called during the diguyScenario::update() call, before anything else is changed due to the update. (That is, characters will not have moved yet, and so on.)
 - void post_update(diguyScenario *s). Called during the diguyScenario::update() call, after everything else is changed due to the update. (That is, characters will have moved, and so on.)

You can use functions such as scenario->get_replaying_history() and scenario->get_t() to branch whether you are replaying history or you are generating new data at runtime. It is up to you to decide what to do for each case.

Update is called once per frame update.

- ♦ void reset(diguyScenario *s). Reset is called whenever the scenario is reset, if the plug-in has its own history, this may be an appropriate time to clear it.
-

i

A key feature of DI-Guy is the ability to save and restore the history of a scenario. Plug-ins can supplement this feature using the review_data functions. The filename argument is the base name of the history review files. It is the programmer's job to write the serialization and deserialization code.

- ♦ Review data functions:
 - void save_review_data(const char* filename). Called when after-action-review data for the scenario is saved by a call to diguyScenario::save_review_data().

Example:

```
save_review_data(const char* filename)
{
    char buffer[MAX_PATH];
    sprintf(buffer, "%s.myplugin", filename);
    FILE * file = fopen(buffer, "w+b");
    assert(file);
    .
    .
    .
    fclose(file);
}
```

- void load_review_data(const char* filename). Called when after-action-review data for the scenario is loaded by a call to diguyScenario::load().
- ♦ Mouse control functions:
 - diguyPluginMouseCursor get_mouse_cursor(diguyScenario* s). Returns the type of cursor that should be used when the mouse is in a 3D View. Please see the documentation for *diguyPluginMouseCursor* for a list of valid values. If this function is not exported from the plug-in, the default cursor DIGUY_PLUGIN_MOUSE_CURSOR_LEFT_ARROW is used.
 - void mouse_func(diguyScenario * s, int x, int y, int down).
 - void mouse_press(diguyScenario* s, diguyPluginMouseEvent* mouse_event). This function is called on an initial press of any mouse button in a 3D View. Use the query functions of the *diguyPluginMouseEvent* object to retrieve which view the press occurred in and the X and Y screen coordinates of the press.
 - void mouse_drag(diguyScenario* s, diguyPluginMouseEvent* mouse_event). Called when the mouse is moved in a 3D View while one or more buttons are pressed.
 - void mouse_release(diguyScenario* s, diguyPluginMouseEvent* mouse_event). Called when any mouse button is released in a 3D View.
 - void mouse_move(diguyScenario* s, diguyPluginMouseEvent* mouse_event). Called when the mouse is moved in a 3D View while no buttons are pressed.

Please see the *diguyApp* documentation for more detailed information on plug-in functions.



7. DI-Guy Author

This chapter explains how you can embed DI-Guy Scenario in other applications.

Introduction	7-2
How DI-Guy Author Works	7-2
How the End User Interacts with Farscene IG	7-2
How DI-Guy Functionality was Added to the IG	7-3
IG Integration	7-5
DI-Guy Author Programming Reference	7-6

7.1. Introduction

DI-Guy Author is a combination of the DI-Guy Scenario application and the DI-Guy Graphics API. It lets you embed the functionality of DI-Guy Scenario in your own application. The 3D parts of DI-Guy Author are 3D library-agnostic, meaning they can be implemented using almost any 3D library.

You subclass the needed 3D visual editing objects, such as, waypoints, path splines, and region meshes, similarly to the way the DI-Guy Graphics API implements the 3D objects needed to render characters in an IG. The API has C++ base classes that provide all of the data needed to create and render editing objects. Your subclasses take that information and render them in your IG.

An additional class, again to be subclassed, provides an interface by which you give DI-Guy input information from your IG such as mouse movement and keystrokes, and by which DI-Guy can ask for needed 3D database information such as line to terrain intersections. With this data DI-Guy can internally modify the scenario. DI-Guy will then tell you when and where to create the 3D visual objects to show the updated state of the scenario.

DI-Guy does all of the calculations for how mouse input should affect waypoint positions and orientations, and therefore path splines, which parts of the terrain should be covered with region polygons painted by the region paintbrush, and other authoring details.

The number of classes needed to be extended, and the amount of information needed to be supplied to DI-Guy, is quite small.

In addition to the extensive editing possible in a 3D environment, DI-Guy Author provides all the dialog boxes and other GUI controls needed to create and manage scenarios.

7.2. How DI-Guy Author Works

To explain how DI-Guy Author works, imagine an application called the FarScene IG that has DI-Guy characters and DI-Guy capabilities. We will describe how the end user interact with the IG and accesses DI-Guy Author through it. Then we will describe what the developer has done under the covers to make this interaction possible. This imaginery implementation of DI-Guy Author is not the only way to do this, just one possible way.

7.2.1. How the End User Interacts with Farscene IG

An end user of the FarScene IG starts the program. In non-author mode the IG runs just as it would if DI-Guy were not present. All visual geometry is rendered by the IG, and all inputs do whatever they would normally do. No DI-Guy elements are visible in either the 3D display or in any dialog boxes or control windows.

DI-Guy Activation

The FarScene IG user decides to edit a scenario using DI-Guy. He selects a menu item that enables Author. A small window opens to show that Author is active and which allows the user to select Author input mode. DI-Guy menus and menu commands are added to the IG interface, similar to those used in DI-Guy Scenario.

Initially, DI-Guy editing is disabled, so other than adding some DI-Guy controls, nothing has really changed in how the application functions.

Opening a Scenario

The user chooses **Author** → **Open Scenario**, and loads a DI-Guy scenario file. The characters and other DI-Guy objects in the scenario are loaded and show up in the 3D window. The user navigates the camera to the scenario location.

The IG user uses the DI-Guy Scenario controls (PeopleBlitzer, CrowdBlitz, and various drawing functions) to add characters and give them actions.

Closing Author

The user exits Author mode, at which point all Author windows and editing visuals in the 3D window disappear. The IG GUI returns to exactly what it looked like before Author was enabled. The characters that were added using DI-Guy Scenario remain in the simulation and move through the IG terrain to act out the edited scenario.

7.2.2. How DI-Guy Functionality was Added to the IG

The FarScene IG user starts it normally. Beyond the fact that DI-Guy Author has been integrated into the IG by a programmer, nothing is different.

DI-Guy Activation

When the user enables Author, a DI-Guy SDK function call gets sent. DI-Guy initializes some internal state to support Author editing. The IG modifies a native UI control, such as a checkbox or menu item, to show that Author is active. However, until the user changes DI-Guy's input mode, the IG still acts as it would normally.

Updating the IG GUI

DI-Guy needs two categories of changes to the IG GUI:

- ♦ The ability to send and receive input in the IG's 3D window so the user can add characters, paths, sensor regions, and so on.
- ♦ Menu commands, dialog boxes, and other GUI controls.

Integrating these two components into the IG requires very different steps.

Native GUI Controls and DI-Guy Author GUI Controls

Native controls refer to controls in the IG GUI that are implemented natively in the IG. While they may query state from DI-Guy and change state appropriately, DI-Guy itself does not create them or change their state. Author controls refer to GUI controls created by and managed by DI-Guy. Author controls show up in subwindows and dialog boxes implemented and controlled by DI-Guy.



Many DI-Guy operations can be accomplished by calling DI-Guy SDK functions based on native UI events. For example, the input mode and playback mode can be set using DI-Guy SDK function calls. This means that if only a small set of DI-Guy operations need to be available, they could be implemented using native controls. If more advanced editing operations are required, the full DI-Guy UI can be enabled.

DI-Guy GUI

While DI-Guy can be used with strictly native controls, there is a lot more user interface available if the Author GUI is enabled. A majority of the GUI in the DI-Guy Scenario application is available in the Author GUI.

Showing Author GUI Windows

Once the Author UI is running, there are DI-Guy function calls that can be made in the IG that will show and hide 2D user interface elements of the Author UI.

Opening a Scenario

The Open Scenario menu item uses DI-Guy function calls in the IG to load a previously created scenario.

PeopleBlitzer

When the user changes the input mode to PeopleBlitzer, the IG is responsible for sending most user input in the 3D window to DI-Guy. This input will mostly include mouse clicks and drags and keyboard activity. When the IG sends the mouse information to DI-Guy while in PeopleBlitzer input mode, DI-Guy internally decides how to handle that information. It looks at its current PeopleBlitzer settings, which include DI-Guy character type, appearance, action, and mind. It then creates the specified character and a path that begins at the clicked point, and ends at the mouse release point. The current character data is shown in the Author GUI so the user can see and edit details of the new character.

3D Editing Visuals

When Author editing is enabled in the 3D window, specialized 3D polygonal objects to represent DI-Guy editing visuals must be shown. These visuals are created by DI-Guy calling virtual functions that were defined when DI-Guy was integrated into the IG. In the PeopleBlitzer case, IG-native geometry for waypoints and paths are requested by DI-Guy. The IG creates these based on recommended geometry and material settings provided by DI-Guy, and positions and orients the visuals from data provided by DI-Guy. When DI-Guy determines that any of these visuals should be repositioned, re-colored, or destroyed, it will notify the IG of this.

CrowdBlitz

When the user selects the CrowdBlitz input mode, mouse inputs will have different effects. When the user is painting regions in the IG 3D window DI-Guy will need to make multiple queries against the IG's native terrain in order to set the points that comprise the region. These queries are made by DI-Guy calling virtual functions defined when DI-Guy was integrated into the IG.

When the user is done and saves the scenario, the current scenario state as it exists in the IG is saved to a *.dss* file.

Closing Author

When the IG user turns off DI-Guy, the Author UI is shut down, and most DI-Guy related native controls and menu items are hidden. The IG should once again run as if DI-Guy were not present.

7.3. IG Integration

If your IG already uses the DI-Guy Graphics API for rendering DI-Guy characters, the integration of DI-Guy Author should be fairly straight-forward.

If your IG does not yet use the DI-Guy Graphics API, that integration must happen first, before you add DI-Guy Author. A DI-Guy Author integration always has to be done in conjunction with a DI-Guy Graphics API integration.

Some of the scenario authoring concepts here will be clearer if you are familiar with DI-Guy Scenario. In short, DI-Guy Scenario is a stand-alone application that can create and edit DI-Guy scenarios interactively. Please see the *DI-Guy Scenario Users Guide* for more information.

7.4. DI-Guy Author Programming Reference

In addition to the information here, there is a substantial amount of documentation in the DI-Guy source code. Most functions in the C++ header files have documentation comments with them. While doing the integration you will likely be referring to the source code documentation often.

In addition, there is an OpenSceneGraph (OSG) based example IG implementation, with source code, to provide concrete examples of how the various functions should be called and classes written. You will probably copy the provided example classes and replace the OSG code with code specific to your 3D environment.

The classes provided by DI-Guy can be broken into two groups: classes for interchanging information between your IG and Author, and classes for rendering 3D editing visuals.

7.4.1. DI-Guy Author Interface Class

The class *diguyAuthorInterface* is the primary way you send and receive information to and from DI-Guy. Its functions can be categorized into the following groups:

- Mouse functions. These functions are called by you from the IG when Author editing is enabled. They provide DI-Guy with the 3D window input such as mouse moves and mouse button clicks it needs for editing the scenario.
- Keyboard function. Similar to mouse functions, but for keystrokes.
- Selected object functions. These functions are called by you from the IG when Author editing is enabled. They tell DI-Guy which object was selected in the 3D window.
- Terrain and intersection virtual functions. These functions are called by DI-Guy when it needs information from the IG about the environment. This includes altitude at a point, converting screen to world coordinates and back, requests to update selected objects, and functions DI-Guy needs for generating its navigation mesh.
- Author UI functions. These functions can be called by you from the IG. They provide control for when the Author UI is launched and request that specific Author UI windows are shown and hidden.
- Author UI status update virtual functions. These functions are called by DI-Guy when the status of various parts of the Author UI application change. This gives the IG a chance to update its state on what is happening with the Author UI, such as enabling or disabling various native user interface controls when a window is opened or closed.

7.4.2. 3D Visual Classes

The API has classes that provide the IG with the information and infrastructure needed to render 3D editing visuals. These must be subclassed by you so you can do things appropriate to your IG. As with the DI-Guy Graphics API, you must register functions for creating instances of your subclasses when DI-Guy needs an object pointer to the base class.

i

These are suggested visual representations of editing visuals. If, for example, you would prefer different colors for editing visuals, you are free to ignore the colors specified by the visual's material. If you would prefer different geometry altogether for waypoints, for example, a textured disc on the ground – you are free to ignore the polygons suggested in the reference implementation.

Most of the geometry visual classes follow the same template, one similar the DI-Guy Graphics API classes. Each of them has most of the following functions:

- ♦ constructor. Called when a new instance of the visual is needed. For example, if a new path is added to the scenario, a new multiline visual will be created. Geometry should not be generated until `build()` is called.
- ♦ `build()`. Called when the geometry for the visual should be generated. It is common for `build()` and `update()` to use a shared utility function for creating polygons, and so on. In the reference implementation this function is called `update_geometry()`.
- ♦ `unbuild()`. Called when geometry created by `build()` should be freed. This is usually, but not always, when the visual is being destroyed.
- ♦ `update()`. Called when the geometry for the visual needs to be updated in some way. This can be because the overall lines or polygons need to be created or recreated, the position of the visual has changed, or the material has changed.

It is common for `build()` and `update()` to use a shared utility function for creating polygons, and so on.

- ♦ `draw()`. Called when the visual should be drawn in the scene. This is usually not necessary for scene-graph style renderers, as they handle drawing themselves.
- ♦ `show()`. Called when the visual should be shown, typically by attaching it to a scene graph or updating a scene graph switch node.
- ♦ `hide()`. Opposite of `show()`.

Visual Materials

Author visual materials are used to specify basic color and other associated information suggested for rendering Author editing visuals. They specify pen color (used for drawing lines), brush color (used for filling polygons), line width, and so on.

Author visuals in general require less information than full DI-Guy Graphics API materials as stored in *diguyGraphicsMaterial* objects. Because of this a “light-weight” class is used that contains only basic information. All Author visuals provide a pointer to an Author visual material.

The class *diguyAuthorMaterial* is used for accessing material information.

Multiline Visuals

Author multiline visuals are used to render path splines. They are a collection of line segments connected end to end. They should be rendered using the visual material’s pen color. As paths become selected and unselected the color of the pen changes.

The class *diguyAuthorVisualMultiline* is used for accessing multiline visual information.

Waypoint Visuals

Author waypoint visuals are used to render path waypoints. Each waypoint is a collection of polygonal solids that show the waypoint’s position and orientation. They should be rendered using the visual material’s brush color. As waypoints become selected and unselected the color of the brush changes.

The class *diguyAuthorVisualWaypoint* is used for accessing waypoint visual information.

Object Handle Visuals

Author object handle visuals are used to render various simple objects that need only a position and orientation. These include path beads, character selection discs, and terrain point pickers. The object handles specify the basic shape the handle should have (disc, sphere, cone, and so on) and the radii, height and other measurements for these shapes. The colors of the object handles are usually dependent on their role.

The class *diguyAuthorVisualObjectHandle* is used for accessing object handle visual information.

Region Mesh Visuals

Author region mesh visuals render the meshes needed for showing the bounds and contents of crowd regions and general regions. They contain the line segments that outline the region boundaries, and the polygons that make up the region interior. In general, the materials for regions are translucent so that stacked regions can be represented.

The class *diguyAuthorVisualRegionMesh* is used for accessing region mesh visual information.

Region Paintbrush Visuals

The Author paintbrush represents the spherical volume that shows where region painting will occur in certain Author input modes. The material for the paintbrush is translucent so that the terrain inside the brush can be seen.

The class *diguyAuthorVisualRegionPaintbrush* is used for accessing paintbrush visual information.



8. Compiling and Linking DI-Guy

Like most SDK products, DI-Guy provides libraries and header files for integrating the product into your application, and programming examples showing how to do so and what the product can do. Please see DI-Guy Release Notes for a list of supported operating systems and compilers.

Compiling and Linking DI-Guy on Windows	8-2
Release and Debug Versions of DI-Guy Libraries	8-2
Lua Script Modules	8-3
Expressive Faces Module for OpenGL	8-3
Programming Examples.....	8-3
Compiling and Linking DI-Guy on Linux.....	8-3
Programming Examples.....	8-4

8.1. Compiling and Linking DI-Guy on Windows

The DI-Guy SDK header files are in *.include*. They are standard C++ header files, and specify the API available to programs using DI-Guy. The full reference for the API is in *DI-Guy SDK Reference Manual*.

The DI-Guy SDK includes several DI-Guy library versions, each compiled with different options. These options determine which instruction set is used, the linking convention, the level of optimization, and whether debugging symbols are present.

Windows libraries are located in *.lib/win/(compiler_version)/*, where *(compiler_version)* is the version of the compiler the library has been compiled for. For example, the library directory for the Windows x86 instruction set, Visual C++ 9, MD runtime library is: *win\x86_vc9_md*.

8.1.1. Release and Debug Versions of DI-Guy Libraries

Visual C++ for Windows has a number of versions of its C Runtime Library. (The Runtime Library provides all of the common C functions such as `printf`, `fopen`, and so on.) The common versions are Multi-threaded Static and Multi-threaded DLL (selected with the `/MD` flag). Both have debug versions. DI-Guy only supports the Multi-threaded DLL versions of the library, or MD and MDd.

Release versions of most libraries and programs are compiled with the MD flag and have optimizations turn on. This is the case for the MD versions of the DI-Guy libraries. Debug versions of most libraries and programs are compiled with the MDd flag, have optimizations turned off, and contain debugging symbols. This is the case for the MDd versions of the DI-Guy libraries.

DI-Guy provides another option that is a hybrid between the common Release and Debug options. Even though a library or program is compiled with the MD flag, meaning it uses the release version of the runtime library, the code that is compiled can still have optimizations turned off and contain debugging symbols. This is the case for the MDn versions of the DI-Guy libraries, when MDn stands for “MD, not optimized”.

This is important because, for stability reasons, all source files and libraries of a given application should be compiled with the same C Runtime Library compiler directives. In other words, if a program will be using the release/MD version of the runtime library, all of the libraries it uses should also use the release/MD version of the Runtime Library. The same is true for the debug/MDd version of the runtime library. (Sometimes MD and MDd versions can co-exist in the same program, but this can sometimes lead to subtle memory corruption bugs.)

Not all third-party libraries are provided with an MDd version, however. In this case DI-Guy provides its MDn version which has been compiled with the `/MD` flag, but is not optimized and contains debugging symbols so that the code can be usefully viewed in a debugging session.

8.1.2. Lua Script Modules

Refer to the *./diguy_ogl/simple_playback* example for an example of using the Lua scripting module.

8.1.3. Expressive Faces Module for OpenGL

Refer to the *./diguy_ogl/facefx* example for an example of using the Expressive Faces module.

8.1.4. Programming Examples

Windows programming example programs and makefiles are in *./programming_examples*. The easiest example program to start with is *./programming_examples/diguy_ogl/simple*.

Each programming example has the code for the example, typically in a *.cpp* file, and a makefile for each compiler that the example can be compiled for. For example, the makefile for the Windows x86 instruction set, Visual C++ 10, MD runtime library is *makefile.win_x86_vc10_md* and can be compiled with *nmake*. A MSVC project file is also available. Similar makefiles are provided for Linux and x64 Windows.

8.2. Compiling and Linking DI-Guy on Linux

The DI-Guy SDK header files are *./include*. They are standard C++ header files, and specify the API available to programs using DI-Guy. The full reference for the API can be found in DI-Guy SDK Reference Manual.

The DI-Guy SDK includes several DI-Guy library versions, each compiled with different options. These options determine which instruction set is used, the level of optimization, whether debugging symbols are present, and whether the library is static or dynamic.

Linux libraries are located in *./lib/linux/(compiler_version)/*, where *(compiler_version)* is the version of the compiler the library has been compiled for. For example, the directory for the linux, i586 instruction set, gcc 4.3, dynamics library is *linux/i586_gcc43_dso*.

8.2.1. Programming Examples

Linux programming example programs and makefiles are in *./programming_examples*.

The easiest example program is to start with is

./programming_examples/diguy_oglsimple.

Each programming example has the code for the example, typically in a *.cpp* file, and a makefile for each compiler that the example can be compiled for. For example, the makefile for the linux, i586 instruction set, gcc 4.3, dynamic library is: *Makefile.linux_i586_gcc43_dso*.

The makefiles are designed for use with GNU make, also called gmake on some systems.



9. Programming Examples

This chapter briefly describes the examples provided with the DI-Guy SDK.

DI-Guy Programming Examples	9-2
OpenGL Examples	9-3

9.1. DI-Guy Programming Examples

The DI-Guy SDK offers many sample programs to help get you started. Examples are in *./programming_examples*.

The contents of the DI-Guy programming examples for Windows is as follows:

OpenGL examples. This is the main programming example directory. It contains examples of working with the DI-Guy OpenGL solution and demonstrates common API features across all rendering platforms.

Graphics API examples. The Graphics API is DI-Guy's open source class-based method for integrating DI-Guy character graphics with various rendering solutions, including OpenGL, OpenSceneGraph, and DirectX 9. The demos show both Immediate Mode (OpenGL), Scene Graph (OpenSceneGraph) style and DirectX 9 rendering, and show how to also use your own geometry loader (not recommended if using skinned characters).

DI-Guy networking. SDK examples of broadcasting and receiving human characters for DIS, HLA 1.3, and HLA 1516. The examples use VR-Link, but are intended to assist developers with networking DI-Guy with other third party and customer proprietary networking solutions.

Plugins. You can create plug-ins, that is, dynamically linked libraries that can be optionally linked to DI-Guy applications at load time. The most typical use is plug-in software that supplements DI-Guy Scenario.

If you install DI-Guy for Vega Prime, software examples will be installed into Vega Prime's resources directory in the *vpDiguy* directory.

DI-Guy digest. The DI-Guy digest is an XML file that describes DI-Guy content in detail, including character types, appearances, and motion information. Users that create custom content can recompile the digest, so that their content will be part of the digest.

9.1.1. OpenGL Examples

OpenGL is DI-Guy's default for programming examples. This means that the *diguy_ogl* directory contains a far more extensive set of examples than the other directories. This does not mean that functionality shown in the OpenGL examples does not work on the other platforms. In fact, the opposite is true. Even if you are not using the DI-Guy OpenGL renderer, do not let that stop you from using the DI-Guy OpenGL examples, which demonstrate lots of DI-Guy functionality that works on all the platforms. There are also a few examples in the *dx9* directory, such as intersection example, that are quite useful.

- ♦ AI Playback. Similar to simple playback, but with DI-Guy AI also activated when the scenario was built in DI-Guy Scenario.
- ♦ AI Populate. Use DI-Guy AI completely through the SDK.
- ♦ Culling. Shows culling of characters based on camera frustum. Includes demonstrates using far positioning.
- ♦ FaceFX. A demonstration of DI-Guy Expressive Faces using the SDK.
- ♦ Gaze. Demonstration of the majority of the Gaze API for commanding characters to look at thing.
- ♦ Get Position. A simple example of querying DI-Guy characters for information.
- ♦ Guide. An excellent demonstration of various DI-Guy guide algorithms in action. A guide is useful when a character's position is being driven intermittently from an external source.
- ♦ Intersection Test. Demonstrates vector intersection to DI-Guy characters.
- ♦ Load Manager. A demonstration of the DI-Guy Load Manager, a method for improving performance in single view DI-Guy applications that manages various LOD-type activities.
- ♦ LODs. Shows DI-Guy Level-of-Detail being activated, including a demonstration using far positioning.
- ♦ Path API. A demonstration of some simple path commands including setting waypoints, action beads, actions, and script beads on a path.
- ♦ Pose. The back joint of a character is rotated directly through the API. In addition, the entire pose of a character is captured and displayed on a second character.
- ♦ Rolling Pitching Ship. DI-Guy characters parented to a moving ship maintaining upright Z posture.
- ♦ Shadow Playback. DI-Guy playback with different shaders.
- ♦ Simple. Our simplest example. Instantiate a DI-Guy soldier character and command him to walk, walk in a crouched posture, go prone, and fire his weapon.
- ♦ Simple Playback. another simple example. Play a *.dss* file as output from DI-Guy Scenario.
- ♦ View Far Settings. Demonstrates fog, light, and camera callbacks. Also shows a scenario running far from the origin, and how to coordinate the camera to avoid jitter caused by lack of precision.

- ♦ View Settings. Demonstrates various methods of setting the camera, fog, and light.



A. Improving Loadtime Performance

This appendix explains how to create a custom configuration file.

Creating a Custom Configuration File	A-2
Using Include Statements in DI-Guy Configuration Files	A-3

A.1. Creating a Custom Configuration File

DI-Guy is configured by default to give you access to all the characters available.

If you need to ship a smaller configuration, you can create a configuration file that culls out elements you do not need. You can specify that your new configuration file be used with a call to `diguy_set_cfg_file()` before calling `diguy_*_initialize()`. We recommend that you put your configuration in `./custom/config`.

For a quick start:

1. Copy `diguy.cfg` from `./config` to `./custom/config/myfastdiguy.cfg`.
2. If you are not using expressive faces, gestures, or motion textures, comment out the lines that refer to them.
3. Change the reference to `diguy_characters.cfg` to be `myfastdiguy_characters.cfg`.
4. Change the reference to `diguy_actors.cfg` to be `myfastdiguy_actors.cfg`.
5. Change the reference to `diguy_appearances.cfg` to be `myfastdiguy_appearances.cfg`.
6. Copy `diguy_characters.cfg` from `./config/diguy` to `./custom/config/diguy/myfastdiguy_characters.cfg`.
7. Delete or comment lines referring to characters that you do not plan to use.
8. Copy `diguy_actors.cfg` from `./config/diguy` to `./custom/config/diguy/myfastdiguy_actors.cfg`.
9. Delete or comment lines referring to actors your characters do not need.
10. Copy `diguy_appearances.cfg` from `./config/diguy` to `./custom/config/diguy/myfastdiguy_appearances.cfg`.
11. Comment our lines referring to appearances you do not need.
12. Change the reference to `appearances_base.cfg` to `myfastappearances_base.cfg`.
13. Copy `appearances_base.cfg` from `./config/diguy` to `./custom/config/diguy/myfastappearances_base.cfg`.
14. Comment out any unused appearances.

A.1.1. Using Include Statements in DI-Guy Configuration Files

DI-Guy behaves as if a preprocessor executed all of the `include` lines before DI-Guy started to read the file. The four `include` directives work as follows:

- ♦ `include file`. Replace the line with the contents of file. Report an error if the file cannot be read.
- ♦ `include_once file`. Replace the line with the contents of file, unless file has already been included. Report an error if file cannot be read.
- ♦ `include_optional file`. Replace the line with the contents of file. Do not report an error if file cannot be read.
- ♦ `include_optional_once file`. Replace the line with the contents of file, unless file has already been included. Do not report an error if file cannot be read.



Index

A

- action
 - bead 2-6
 - character 2-6
 - closest 2-14
 - direction 2-10
 - distance 2-10
 - facing angle 2-10
 - forcing 2-16
 - functions 2-16
 - getting 2-10
 - guide 2-22
 - mode 2-8
 - setting, in pipeline 1-7
 - speed 2-10
- action mode 2-16
- add_callback() function 2-13, 2-19
- add_default_character_callback() function 2-13
- add_guide() function 2-22
- add_view_camera() function 4-4
- advancing, time 2-9, 2-15
- after action review, AAR 2-13
- Aim, API 3-5
- aim, functions 2-19
- aiming, weapon 3-5
- altitude, functions 2-20
- animation, joint 3-2
- animation blend tree 2-17
- API
 - Aim 3-5
 - Gaze and Pointing 3-4
 - graphics 2-3, 2-23
 - multithreaded 2-10

- appearance
 - functions 2-15
 - getting 2-10
- attribute, custom 5-12
- Author, DI-Guy 7-2

B

- bead, action 2-6
- blend tree, animation 2-17
- boundary, zone 4-4
- bounding radius, character 2-16
- bounding sphere 2-9
- build() function 7-7

C

- callback
 - CALLBACK_ID_CURRENT_ACTION_CHANGED() 1-7
 - character 2-13
 - functions 2-19
 - default 2-13
 - functions 2-13
 - ID 2-13, 2-19
 - mapping to event handler 2-13
- CALLBACK_ID_CURRENT_ACTION_CHANGED(), callback 1-7
- camera 2-7
 - functions 2-12
 - view 2-12
 - zone boundary 4-4
- camera_changed() function 4-4
- change_view_camera_settings() function 4-4
- channel, gesture 3-6

- character 2-6
 - action 2-6
 - bounding radius 2-16
 - callbacks 2-13
 - creating 2-5, 2-10
 - creating and controlling 1-4
 - culling 2-16
 - death 2-15
 - destroying 2-10
 - functions 2-10
 - geographic position and orientation 2-14
 - group 2-14
 - head appearance 2-15
 - joint animation 3-2
 - LOD, functions 2-17
 - name 2-14
 - parent 2-14
 - parenting, function 2-17
 - position 2-6
 - resuscitating 2-15
 - scale 2-14
 - scenario in 2-14
 - sensing presence of 2-11
 - speed 2-16
 - status 2-14
 - time range 2-14
 - type 2-14
 - updating 2-5
 - user data function 2-17
 - user variables 2-22
 - when drawn 2-14
- character mode
 - free 1-7
 - path 1-7
- character type, getting 2-10
- Character Viewer, starting 2-2
- CIGI 1-5
- class
 - diguyApp* 2-5, 2-7, 2-10, 4-4
 - diguyAuthorInterface* 7-6
 - diguyAuthorMaterial* 7-8
 - diguyAuthorVisualMultiline* 7-8
 - diguyAuthorVisualObjectHandle* 7-8
 - diguyAuthorVisualRegionMesh* 7-8
 - diguyAuthorVisualRegionPaintbrush* 7-9
 - diguyAuthorVisualWaypoint* 7-8
 - diguyCharacter* 2-6, 2-9, 2-14, 2-18, 4-3
 - diguyCharacterPath* 2-8
 - diguyGraphicsLink* 2-6
 - diguyGraphicsMaterial* 7-8
 - class, continued*
 - diguyGraphicsShaderTechnique* 4-5
 - diguyGraphicsShape* 2-6
 - diguyImpact* 2-20, 5-14
 - diguyLoadManager* 4-4
 - diguyNetInterface* 5-3, 5-4, 5-10
 - diguyPaths* 2-6
 - diguyPathShape* 2-6, 2-8
 - diguyPluginMouseCursor* 6-4
 - diguyPluginMouseEvent* 6-4
 - diguyScenario* 2-5, 2-6, 2-7, 2-9, 2-10, 2-11, 2-12
 - diguySceneObject* 2-7
 - diguyView* 2-7
 - diguyWaypoint* 2-6
- closest action 2-14
- compiling
 - Linux 8-3
 - Windows 8-2
- compressing, geometry and textures 4-6
- configuration
 - network, parameter 5-15
- configuration file
 - diguy.cfg* A-2
 - network_model_map.cfg* 5-6
 - network.cfg* 5-4, 5-9, 5-15
- configuring, HLA 5-8
- coordinate
 - converting 5-18
 - geodetic 5-18
- create function 2-7
- create_and_force_bridge_path()* function 2-18
- create_character()* function 2-5, 2-7, 2-10, 4-6
- create_gesture()* function 4-6
- create_guide()* function 2-22
- create_path()* function 2-18
- create_simple_path()* function 2-18
- create_sound_instance()* function 2-11
- creating
 - character 2-10
 - characters 2-5
- culling
 - character 2-16
 - draw 4-2
 - update 4-2, 4-4
- custom
 - attribute 5-12
 - FOM 5-2
 - PDU 5-2, 5-4

D

- database, coordinates 5-18
- dead-reckoning 2-22
- death, character 2-15
- debug, libraries 8-2
- degrees of freedom, joint 3-2
- deinitialization
 - functions, DI-Guy 2-3
- deinitializing, modules 2-4
- delete function 2-7
- destroy_character() function 2-10
- destroying, character 2-10
- detonation
 - interaction 5-13
 - PDU 5-13
- die_now() function 2-15
- DI-Guy
 - Author 7-2
 - FOM 1-5
 - networking with simulations 1-5
- DI-Guy AI 2-4, 2-5, 2-8
- DI-Guy Character Viewer 2-16, 4-3
- DI-Guy Graphics API 7-5
- DI-Guy Lifeform Server 5-2
- DI-Guy Load Manager 4-3, 4-4
- DI-Guy Scenario 1-3, 5-2
 - embedding in IG 7-2
 - Gesture Previewer 2-21
 - information popups 2-12
 - network model map application 5-6
 - plug-ins for 6-2
 - remote control 5-19
- diguy_ai_deinitialize() function 2-4
- diguy_ai_initialize() function 2-4
- diguy_create_scenario() function 2-5
- diguy_deinitialize() function 2-3
- diguy_dis_constants.h* header file 5-5
- diguy_facefx_deinitialize() function 2-4
- diguy_facefx_initialize() function 2-4
- diguy_graphics_api_initialize() function 1-3
- diguy_graphics_initialize() function 2-3
- diguy_module_net_deinitialize() function 2-4, 5-3
- diguy_module_net_initialize() function 2-4
- diguy_ogl_initialize() function 2-3
- diguy_script_lua_deinitialize() function 2-4
- diguy_script_lua_initialize() function 2-4
- diguy_set_cfg_file() function A-2
- diguy_sound_openal_deinitialize() function 2-4
- diguy_sound_openal_initialize() function 2-4
- diguyApp* class 2-5, 2-7, 2-10, 4-4
- diguyAuthorInterface* class 7-6
- diguyAuthorMaterial* class 7-8
- diguyAuthorVisualMultiline* class 7-8
- diguyAuthorVisualObjectHandle* class 7-8
- diguyAuthorVisualRegionMesh* class 7-8
- diguyAuthorVisualRegionPaintbrush* class 7-9
- diguyAuthorVisualWaypoint* class 7-8
- diguy.cfg, configuration file A-2
- diguyCharacter* class 2-6, 2-9, 2-14, 2-18, 4-3
- diguyCharacter.h* header file 2-17, 2-19
- diguyCharacterPath* class 2-8
- diguyGraphicsLink* class 2-6
- diguyGraphicsMaterial* class 7-8
- diguyGraphicsShaderTechnique* class 4-5
- diguyGraphicsShape* class 2-6
- diguyImpact* class 2-20, 5-14
- diguyLoadManager* class 4-4
- diguyNetInterface* class 5-3, 5-4, 5-10
- diguyPaths* class 2-6
- diguyPathShape* class 2-6, 2-8
- diguyPluginMouseCursor* class 6-4
- diguyPluginMouseEvent* class 6-4
- diguyScenario* class 2-5, 2-6, 2-7, 2-9, 2-10, 2-11, 2-12
- diguyScenario.h* header file 2-13
- diguySceneObject* class 2-7
- diguyView* class 2-7
- diguyWaypoint* class 2-6
- DirectX 2-23
- DIS 1-5, 2-4
 - entity type 5-6
 - example 1-6
 - functions 2-21
 - networking 5-3
 - update rate 5-4
- DLL 6-2
- draw culling 4-2
- draw_pass1() function 2-9, 2-15
- draw_pass2() function 2-9, 2-15
- draw() function 2-5, 2-9, 2-15, 7-7
- DSS file 1-3, 2-9
- DXT5_nm format 4-6
- dying, character 2-15

E

- embedding, scenario 1-3
- entity, mapping 5-6
- entity mapping, HLA 5-11

- entity type, DIS 5-6
- environment variable
 - RTI_CONFIG 5-9
 - RTI_RID_FILE 5-8
- event
 - handler 2-11
 - script 2-12
- event handler
 - functions 2-19
 - mapping to callback 2-13
- example
 - DIS 1-6
 - guide_example 1-8
 - HLA 1-6
 - net_rcv 5-3
 - net_send 5-3
 - programs 8-3, 8-4
 - list of 9-2
 - simple 1-4
 - simple_playback 1-3
- execute_1stage_gesture() function 4-6
- execute_2stage_gesture() function 4-6
- execute_3stage_gesture() function 4-6
- execute_gesture() function 4-6
- exercise, setting location 5-19
- Expressive Faces 2-4, 2-12, 8-3

F

- facial expression, functions 2-12
- FDD, file 5-9
- FED, file 5-9
- file
 - DSS 1-3, 2-9
 - FDD 5-9
 - FED 5-9
- find_link() function 2-17
- fire
 - interaction 5-13
 - PDU's 5-13
- fire_weapon_n_times() function 2-20
- fire_weapon() function 2-20
- flat earth 5-18
- fog, functions 2-12
- FOM
 - custom 5-2
 - DI-Guy 1-5
- force 2-11
- force_action_and_path_shape() function 2-16
- force_action_duration() function 2-16

- force_action() function 1-8, 2-16, 2-18
- force_partial_path() function 2-18
- force_path() function 2-18
- forcing, action 2-16
- free
 - action mode and position mode 2-8
 - character mode 1-7
- Free Action Mode 2-16
- free action mode 2-22
- free mode 2-6
- Free Position Mode 2-16, 2-18
- free position mode 2-22
- friendly, group 2-11
- frustum 2-9
- function
 - action 2-16
 - add_callback() 2-13, 2-19
 - add_default_character_callback() 2-13
 - add_guide() 2-22
 - add_view_camera() 4-4
 - aim 2-19
 - altitude 2-20
 - appearance 2-15
 - build() 7-7
 - callback 2-13
 - character 2-19
 - camera 2-12
 - camera_changed() 4-4
 - change_view_camera_settings() 4-4
 - character 2-10
 - LOD 2-17
 - parenting 2-17
 - create 2-7
 - create_and_force_bridge_path() 2-18
 - create_character() 2-5, 2-7, 2-10, 4-6
 - create_gesture() 4-6
 - create_guide() 2-22
 - create_path() 2-18
 - create_simple_path() 2-18
 - create_sound_instance() 2-11
 - delete 2-7
 - destroy_character() 2-10
 - die_now() 2-15
 - diguy_ai_deinitialize() 2-4
 - diguy_ai_initialize() 2-4
 - diguy_create_scenario() 2-5
 - diguy_deinitialize() 2-3
 - diguy_facefx_deinitialize() 2-4
 - diguy_facefx_initialize() 2-4
 - diguy_graphics_api_initialize() 1-3

function, continued

- diguy_graphics_initialize() 2-3
- diguy_module_net_deinitialize() 2-4, 5-3
- diguy_module_net_initialize() 2-4
- diguy_ogl_initialize() 2-3
- diguy_script_lua_deinitialize() 2-4
- diguy_script_lua_initialize() 2-4
- diguy_set_cfg_file() A-2
- diguy_sound_openal_deinitialize() 2-4
- diguy_sound_openal_initialize() 2-4
- DIS 2-21
- draw_pass1() 2-9, 2-15
- draw_pass2() 2-9, 2-15
- draw() 2-5, 2-9, 2-15, 7-7
- event handler 2-19
- execute_1stage_gesture() 4-6
- execute_2stage_gesture() 4-6
- execute_3stage_gesture() 4-6
- execute_gesture() 4-6
- facial expression 2-12
- find_link() 2-17
- fire_weapon_n_times() 2-20
- fire_weapon() 2-20
- fog 2-12
- force_action_and_path_shape() 2-16
- force_action_duration() 2-16
- force_action() 1-8, 2-16, 2-18
- force_partial_path() 2-18
- force_path() 2-18
- gaze 2-19
- gesture 2-21
- get_action_mode() 2-16
- get_appearance() 2-15
- get_base_link() 2-17
- get_bounding_radius() 2-16
- get_current_action() 1-7
- get_current_appearance() 2-15
- get_current_hand_item() 2-15
- get_current_head_appearance() 2-15
- get_current_tin() 2-14
- get_current_tout() 2-14
- get_dead() 2-15
- get_default_bounding_radius() 2-16
- get_desired_action() 1-7, 2-16
- get_drawn_by_scenario_flag() 2-14
- get_enabled() 2-14
- get_graphics_lod() 2-16
- get_last_impact_record() 2-20
- get_link_at_index() 2-17
- get_link_position_double() 2-17

function, continued

- get_link_position_with_offset_double() 2-17
- get_link_position_with_offset() 2-17
- get_link_position() 2-17
- get_link_relative_position() 2-17
- get_lod_ranges() 2-17
- get_max_ticks_behind_till_dropped() 2-10
- get_motion_lod() 2-17
- get_name() 2-14
- get_num_characters() 2-10
- get_orientation() 2-14
- get_playback_loop() 2-9
- get_position_double() 2-14
- get_position_mode() 2-18
- get_position() 2-14
- get_scale() 2-14
- get_scenario() 2-14
- get_shader_lod() 2-16
- get_speed() 2-16
- get_t_controlled_by_scenario_t() 2-15
- get_t() 2-9
- get_tick_dt() 2-9
- get_ticks_can_be_dropped() 2-10
- get_time_to_transition() 2-16
- get_tout() 2-9
- get_type_name() 2-14
- get_up_vector() 2-16
- get_user_data() 2-17
- group 2-11
- guide 2-22
- hide() 7-7
- history 2-13
- initialization and deinitialization, DI-Guy 2-3
- initialize 4-6
- is_group_member() 2-14
- is_within_distance_n_of_character() 2-19
- is_within_distance_n_of_member_of_group() 2-19
- jump_to_action_bead() 2-18
- leave_path() 2-18
- light 2-12
- link 2-17
- load_camera_settings() 2-12
- load() 2-5, 2-9
- LOD 2-13, 2-17
- map_event_handler_to_callback_id() 2-13, 2-19
- meta-action 2-14
- motion LOD 2-17
- move_camera() 4-4

function, continued

- new 2-7
- nod 2-19
- path 2-18
- perception 2-19
- performance 2-10
- play_3d_sound() 2-18
- play_sound() 2-11, 2-18
- playback control 2-9
- point 2-19
- preload 2-10
- preload_appearance() 2-10, 4-6
- preload_character_type() 2-10, 4-6
- preload_gesture() 2-10, 4-6
- push_path() 2-18
- remove_all_guides() 2-22
- remove_guide() 2-22
- reset_signals() 2-11
- reset() 2-9
- resume_interrupted_path() 2-18
- review 2-13
- revive_now() 2-15
- scenario management 2-5
- scenario_update() 1-4
- set_animation_velocity() 2-17
- set_appearance() 2-15
- set_bounding_radius() 2-16
- set_current_appearance() 2-15, 4-6
- set_current_head_appearance() 2-15
- set_current_tin() 2-14
- set_current_tout() 2-14
- set_default_bounding_radius() 2-16
- set_default_character_altitude_function() 2-13
- set_desired_action() 1-7, 2-16, 2-18
- set_desired_orientation() 2-22
- set_desired_position() 2-22
- set_desired_velocity() 2-22
- set_distance_along_path() 2-18
- set_drawn_by_scenario_flag() 2-14
- set_enabled() 2-14
- set_graphics_lod() 2-16, 2-17, 4-3
- set_guide_algorithm_float_parameter() 2-22
- set_hand_item() 2-15
- set_history_type() 2-13
- set_iguy_character() 2-10
- set_initial_orientation() 2-14
- set_initial_position() 2-14
- set_lod_ranges() 2-17
- set_max_ticks_behind_till_dropped() 2-10
- set_max_update_period() 4-5

function, continued

- set_motion_lod() 2-17
- set_name() 2-14
- set_num_threads() 2-10
- set_orientation_relative_to_parent() 2-14
- set_orientation() 2-14
- set_parent() 2-17
- set_playback_loop() 2-9
- set_playback_loop() function 2-9
- set_position_double() 2-14
- set_position_relative_to_parent() 2-14
- set_position() 2-14, 2-20
- set_scale() 2-14
- set_shader_lod() 2-16
- set_speed() 2-16
- set_t_controlled_by_scenario_t() 2-15
- set_t_scale_factor() 2-16
- set_ticks_can_be_dropped() 2-10
- set_timeout_interval() 5-4
- set_tout() 2-9
- set_type_name() 2-14
- set_up_vector() 2-16
- set_user_data() 2-17
- set_weapon_fires_live_rounds() 2-20
- set_zone_bounds() 4-4
- set_zone_parameters() 4-5
- show() 7-7
- sound 2-11, 2-18
- sound instance 2-18
- speed 2-16
- stop_all_sounds() 2-11
- time 2-15
- time control 2-9
- time-of-day 2-9
- unbuild() 7-7
- unset_parent() 2-17
- unset_t_scale_factor() 2-16
- update_geometry() 7-7
- update() 2-5, 2-9, 2-15, 7-7
- user data 2-17
- user variable 2-12, 2-22
- weapon 2-20

G

- gaze, functions 2-19
- Gaze and Pointing, API 3-4
- gazing 3-4
- geocentric 5-18
- geodetic, coordinate 5-18

geometry, compression 4-6
 gesture 3-6
 channel 3-6
 functions 2-21
 getting 2-10
 Gesture Previewer, DI-Guy Scenario 2-21
 get_action_mode() function 2-16
 get_animation_angular_velocity header file 2-17
 get_animation_local_velocity header file 2-17
 get_animation_velocity() header file 2-17
 get_appearance() function 2-15
 get_base_link() function 2-17
 get_bounding_radius() function 2-16
 get_current_action() function 1-7
 get_current_appearance() function 2-15
 get_current_hand_item() function 2-15
 get_current_head_appearance() function 2-15
 get_current_tin() function 2-14
 get_current_tout() function 2-14
 get_dead() function 2-15
 get_default_bounding_radius() function 2-16
 get_desired_action() function 1-7, 2-16
 get_drawn_by_scenario_flag() function 2-14
 get_enabled() function 2-14
 get_graphics_lod() function 2-16
 get_last_impact_record() function 2-20
 get_link_at_index() function 2-17
 get_link_position_double() function 2-17
 get_link_position_with_offset_double() function 2-17
 get_link_position_with_offset() function 2-17
 get_link_position() function 2-17
 get_link_relative_position() function 2-17
 get_lod_ranges() function 2-17
 get_max_ticks_behind_till_dropped() function 2-10
 get_motion_lod() function 2-17
 get_name() function 2-14
 get_num_characters() function 2-10
 get_orientation() function 2-14
 get_playback_loop() function 2-9
 get_position_double() function 2-14
 get_position_mode() function 2-18
 get_position() function 2-14
 get_scale() function 2-14
 get_scenario() function 2-14
 get_shader_lod() function 2-16
 get_speed() function 2-16
 get_t_controlled_by_scenario_t() function 2-15
 get_t() function 2-9

get_tick_dt() function 2-9
 get_ticks_can_be_dropped() function 2-10
 get_time_to_transition() function 2-16
 get_tout() function 2-9
 get_type_name() function 2-14
 get_up_vector() function 2-16
 get_user_data() function 2-17
 graphic
 LOD 4-2
 LODs 2-13
 graphics, API 2-3, 2-23
 group
 character 2-14
 functions 2-11
 guide, functions 2-22
 guide_example, example 1-8

H

head, nodding and shaking 3-6
 head appearance, character 2-15
 header file
 diguy_dis_constants.h 5-5
 diguyCharacter.h 2-17, 2-19
 diguyScenario.h 2-13
 get_animation_angular_velocity 2-17
 get_animation_local_velocity 2-17
 get_animation_velocity() 2-17
 set_animation_angular_velocity 2-17
 set_animation_local_velocity 2-17
 hide() function 7-7
 High Level Architecture 5-7
 history, functions 2-13
 HLA 1-5, 2-4, 5-7
 configuring 5-8
 entity mapping 5-11
 example 1-6
 troubleshooting 5-13
 update rate 5-11

I

ID, callback 2-13, 2-19
 IG 1-8
 embedding DI-Guy Scenario 7-2
 I-Guy 2-10
 immediate mode 2-5
 rendering system 2-9
 include statement A-3
 information, popups 2-12

initialize, functions 4-6
initializing, modules 2-4
interaction, fire and detonate 5-13
is_group_member() function 2-14
is_within_distance_n_of_character() function 2-19
is_within_distance_n_of_member_of_group() function 2-19

J

join, updating 2-9
joint 2-3, 2-6
 animation 3-2
joystick 2-10
jump_to_action_bead() function 2-18

L

leave_path() function 2-18
library, release and debug 8-2
light, functions 2-12
link 2-3, 2-6
 functions 2-17
linking
 Linux 8-3
 Windows 8-2
Linux, compiling and linking 8-3
load management 4-4
Load Manager 2-7, 4-4
load_camera_settings() function 2-12
load() function 2-5, 2-9
loading, performance A-2
location
 character 2-14
 exercise, setting 5-19
LOD 4-2
 functions 2-13, 2-17
 graphic 2-13
 motion 2-13, 4-4, 4-5
 shader 4-3
Lua 2-4
 module 8-3

M

m_bdimessageid 5-5
m_bdimessageversion 5-5
m_data 5-5

m_datasize 5-5
m_entityID 5-5
m_pdu_header 5-5
map_event_handler_to_callback_id() function 2-13, 2-19
mapping, entities 5-6
mesh 2-6
meta-action, functions 2-14
MLOD 4-5
mode
 action 2-8, 2-16
 free 2-6
 immediate 2-5
 path 2-6
 position 2-8, 2-16
module, initializing and deinitializing 2-4
motion, LODs 2-13
motion level of detail 4-5
motion LOD 4-4
 function 2-17
motion pipeline 3-2
move_camera() function 4-4
multi-threaded control API 2-10
multithreading 4-3

N

name, character 2-14
net_recv, example 5-3
net_send, example 5-3
network, configuration parameter 5-15
network model map application, DI-Guy Scenario 5-6
network_model_map.cfg, configuration file 5-6
network.cfg, configuration file 5-4, 5-9, 5-15
networking 5-2
 DIS 5-3
 HLA, enabling 5-10
 simulations and DI-Guy 1-5
neutral, group 2-11
new function 2-7
nod, functions 2-19
nodding, head 3-6
normal map, texture 4-6

O

object, scene 2-7, 2-12
OpenAL 2-4
OpenGL 1-3, 2-3, 2-23

OpenSceneGraph 2-23, 7-6

opposing, group 2-11

orientation

character 2-14

guide 2-22

relative to parent 2-14

overriding, user pose 3-5

P

parameter

network configuration 5-15

zone 4-5

parent, character 2-14

path

action mode and position mode 2-8

character mode 1-7

functions 2-18

mode 2-6

shape 2-6, 2-11

Path Action Mode 2-16

Path Position Mode 2-16, 2-18

path shape 2-11, 2-16

PDU

custom 5-2, 5-4

detonation 5-13

fire 5-13

perception, functions 2-19

performance 4-2

functions 2-10

loading A-2

pipeline

motion 3-2

querying 1-7

setting actions 1-7

play_3d_sound() function 2-18

play_sound() function 2-11, 2-18

playback, functions 2-9

plug-in, writing 6-2

point, functions 2-19

pointing 3-4

popup, information 2-12

pose

user, overriding 3-5

position

character 2-6

geographic 2-14

guide 2-22

mode 2-8

updating 2-9

position mode 2-16

preload, functions 2-10

preload_appearance() function 2-10, 4-6

preload_character_type() function 2-10, 4-6

preload_gesture() function 2-10, 4-6

preloading 4-6

program

example 8-3, 8-4

list of 9-2

push_path() function 2-18

Q

querying, pipeline 1-7

R

region, sensor 2-11

release, libraries 8-2

remote control, DI-Guy Scenario 5-19

remove_all_guides() function 2-22

remove_guide() function 2-22

renderer

immediate mode 2-5

scene graph 2-5

rendering, window 2-12

rendering system, immediate mode 2-9

reset_signals() function 2-11

reset() function 2-9

resume_interrupted_path() function 2-18

resuscitating, character 2-15

review, functions 2-13

revive_now() function 2-15

rewinding, scenario 2-9

rotation_threshold 5-4

route 2-11

RPR FOM 5-7

RTI 5-7

RTI_CONFIG, environment variable 5-9

RTI_RID_FILE, environment variable 5-8

S

scale, character 2-14

scaling, time 2-16

scenario 1-3, 2-6

character is in 2-14

DSS file 2-9

embedding 1-3

- scenario 1-3, 2-6, continued
 - group 2-11
 - management, function 2-5
 - rewinding 2-9
- scenario_update() function 1-4
- scene, object 2-12
- scene graph, renderer 2-5
- scene object 2-7
- script, event 2-12
- scripting, Lua 8-3
- sensor, region 2-11
- set_animation_angular_velocity* header file 2-17
- set_animation_local_velocity* header file 2-17
- set_animation_velocity() function 2-17
- set_appearance() function 2-15
- set_bounding_radius() function 2-16
- set_current_appearance() function 2-15, 4-6
- set_current_head_appearance() function 2-15
- set_current_tin() function 2-14
- set_current_tout() function 2-14
- set_default_bounding_radius() function 2-16
- set_default_character_altitude_function()
 - function 2-13
- set_desired_action() function 1-7, 2-16, 2-18
- set_desired_orientation() function 2-22
- set_desired_position() function 2-22
- set_desired_velocity() function 2-22
- set_distance_along_path() function 2-18
- set_drawn_by_scenario_flag() function 2-14
- set_enabled() function 2-14
- set_graphics_lod() function 2-16, 2-17, 4-3
- set_guide_algorithm_float_parameter() function
 - 2-22
- set_hand_item() function 2-15
- set_history_type() function 2-13
- set_iguy_character() function 2-10
- set_initial_orientation() function 2-14
- set_initial_position() function 2-14
- set_lod_ranges() function 2-17
- set_max_ticks_behind_till_dropped() function 2-10
- set_max_update_period() function 4-5
- set_motion_lod() function 2-17
- set_name() function 2-14
- set_num_threads() function 2-10
- set_orientation_relative_to_parent() function 2-14
- set_orientation() function 2-14
- set_parent() function 2-17
- set_playback_loop() function 2-9

- set_position_double() function 2-14
- set_position_relative_to_parent() function 2-14
- set_position() function 2-14, 2-20
- set_scale() function 2-14
- set_shader_lod() function 2-16
- set_speed() function 2-16
- set_t_controlled_by_scenario_t() function 2-15
- set_t_scale_factor() function 2-16
- set_ticks_can_be_dropped() function 2-10
- set_timeout_interval() function 5-4
- set_tout() function 2-9
- set_type_name() function 2-14
- set_up_vector() function 2-16
- set_user_data() function 2-17
- set_weapon_fires_live_rounds() function 2-20
- set_zone_bounds() function 4-4
- set_zone_parameters() function 4-5
- setting
 - action, in pipeline 1-7
 - exercise location 5-19
- shader, LODs 4-3
- shaking, head 3-6
- shape 2-6
 - path 2-6, 2-11
- show() function 7-7
- signal 2-11
- simple, example 1-4
- simple_playback, example 1-3
- skeletal, structure 2-3
- sound
 - functions 2-11, 2-18
 - initializing and deinitializing 2-4
- sound instance, functions 2-18
- speed
 - character 2-16
 - functions 2-16
- spline 2-11
- starting, Character Viewer 2-2
- status, character 2-14
- stop_all_sounds() function 2-11
- structure, skeletal 2-3

T

- terrain 2-12
- texture
 - compression 4-6
 - DXT5_nm 4-6
 - normal map 4-6
- threading, multi 4-3

threshold, timeout 5-4
time
 advancing 2-9, 2-15
 functions 2-9, 2-15
 scaling 2-16
time range, character 2-14
time-of-day, functions 2-9
timeout, threshold 5-4
translation_threshold 5-4
troubleshooting, HLA 5-13
type, character 2-14

U

unbuild() function 7-7
uninitialization
 functions, DI-Guy 2-3
unset_parent() function 2-17
unset_t_scale_factor() function 2-16
up vector 2-16
update culling 4-2, 4-4
update rate
 DIS 5-4
 HLA 5-11
update_geometry() function 7-7
update() function 2-5, 2-9, 2-15, 7-7
updating
 character 2-5
 joints 2-9
 position 2-9
user data, functions 2-17
user pose, overriding 3-5
user variable, functions 2-12, 2-22
UTM 5-18
 converting to geodetic 5-19

V

variable, user 2-22
vector, up or Z 2-16
Vega Prime 2-3, 9-2
view 2-7
 window 2-12
view frustum 4-2
VR-Link 1-6, 5-2

W

waypoint 2-6, 2-11, 2-12

weapon
 aiming 3-5
 functions 2-20
WGS84 5-18
window
 rendering 2-12
 view 2-12
Windows, compiling and linking 8-2
writing, plug-in 6-2

X-Y-Z

Z vector 2-16
zone
 boundary 4-4
 parameter 4-5



Link - Simulate - Visualize

DIG-13.0-7-140623