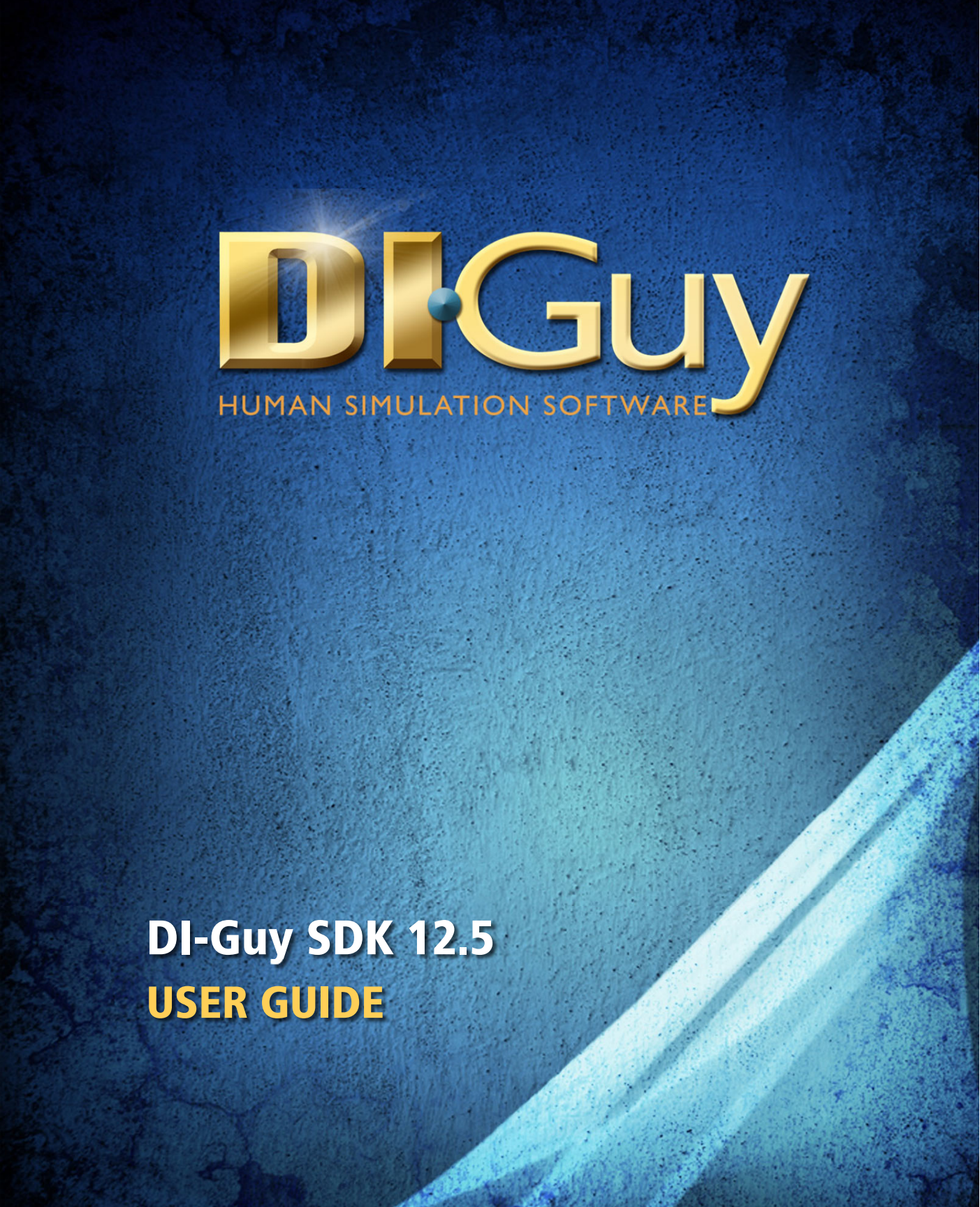

DI-Guy
HUMAN SIMULATION SOFTWARE
DI-Guy SDK 12.5
USER GUIDE

DI-Guy SDK User Guide

Version 12.5

Boston Dynamics
78 Fourth Avenue
Waltham, MA 02451 USA

For questions regarding *DI-Guy SDK*,
send email to diguy@diguy.com.

Part No. DI-Guy SDK 12.5-User Guide

DI-Guy Software License Agreement

This is a legally binding agreement between you ("LICENSEE") and Boston Dynamics, with its principal place of business at 78 Fourth Avenue Waltham, MA USA ("Boston Dynamics"). By breaking the seal on the package containing the DI-Guy products (DI-Guy SDK, DI-Guy Scenario, DI-Guy AI and DI-Guy Motion Editor), and/or by installing and/or using the enclosed software, data, models, documentation, or related recorded information, regardless of its form or the method of the recording (the "SOFTWARE"), LICENSEE is agreeing to be bound by the terms and conditions of this agreement, including the software license and disclaimer of software warranty below. Please read this document carefully before opening the package and using the SOFTWARE. If LICENSEE does not agree with the terms and conditions of this agreement, LICENSEE should promptly return the unopened package and the SOFTWARE to the place where it was obtained, and LICENSEE will be given a full refund of any license fee that LICENSEE paid.

1. Grant of License: Subject to the terms and conditions of this Agreement, Boston Dynamics hereby grants to LICENSEE a non-transferable and non-exclusive right to use and execute the SOFTWARE on a single computer system at one time. LICENSEE agrees that it will not reverse engineer, decompile or disassemble any portion of the SOFTWARE, nor extract data of any kind from the SOFTWARE, including but not limited to motion data, 3D models, textures, texture movies, image sequences, terrain databases and the like. If LICENSEE disposes of any media or apparatus containing SOFTWARE, LICENSEE will ensure that it has completely erased or otherwise destroyed any SOFTWARE contained on such media or stored in such apparatus. LICENSEE may not distribute, lease, transfer, export, loan or otherwise convey the SOFTWARE or any portion thereof to anyone.

2. Copying Restrictions: In order to effect LICENSEE's license rights hereunder, it may install the SOFTWARE by copying it onto the hard disk drive or into the CPU memory of a computer system for use thereon, and may make full or partial copies of the SOFTWARE, but only as necessary for backup or archival purposes. LICENSEE agrees that (i) LICENSEE's use and possession of such copies shall be solely under the terms and conditions of this Agreement, and (ii) LICENSEE shall place the same proprietary and copyright notices and legends on all such copies as included by Boston Dynamics on the media containing the authorized copy of the SOFTWARE originally provided by Boston Dynamics.

3. Ownership: LICENSEE agrees and acknowledges that Boston Dynamics transfers no ownership interest in the SOFTWARE, in the intellectual property in SOFTWARE, nor in any SOFTWARE copy to LICENSEE under this Agreement or otherwise, and that Boston Dynamics and its licensors reserve all rights not expressly granted hereunder. After LICENSEE pays any applicable license fees, LICENSEE will own the media on which the SOFTWARE was originally provided hereunder and on which LICENSEE subsequently copies the SOFTWARE, but Boston Dynamics and its licensors shall retain ownership of all SOFTWARE and copies of the SOFTWARE or portions thereof embodied in or on any media.

4. Transfer Restrictions: LICENSEE may not sublicense, transfer, or assign this Agreement or any rights or obligations under this Agreement, in whole or in part.

5. Export Restrictions: LICENSEE agrees not to export or re-export, directly or indirectly, from the United States through any country or to any country of ultimate destination defined by the United States Government or any agency thereof as an "Excluded Territory". LICENSEE will not export, directly or indirectly, the Licensed Programs or Documentation to any country from which the United States Government or any agency thereof requires an export license or other governmental approval at the time of export without first obtaining such license or approval. The U.S. Government's list of "Excluded Territories" is subject to change without notice and is therefore not included herein.

6. Confidentiality: By accepting this license, you acknowledge that the SOFTWARE and accompanying materials, and any proprietary information contained in the media, are proprietary in nature and contain valuable confidential information developed or acquired at great expense. You agree not to disclose to others or utilize such trade secrets or proprietary information except as provide herein.

7. Enforcement of Terms: If LICENSEE fails to fulfill any of its obligations under this Agreement, Boston Dynamics and/or its licensors may pursue all available legal remedies to enforce this Agreement, and Boston Dynamics may, at any time after LICENSEE's default of this Agreement, terminate this Agreement and all licenses and rights granted under this Agreement. LICENSEE agrees that Boston Dynamics' licensors referenced in the SOFTWARE are third-party beneficiaries of this Agreement, and may enforce this Agreement as it relates to their intellectual property. LICENSEE further agrees that, if Boston Dynamics terminates this Agreement for default, LICENSEE will, within ten (10) days after any such termination, deliver to Boston Dynamics or render unusable all SOFTWARE originally provided hereunder and any copies thereof embodied in any medium.

8. Governing Law: This Agreement shall be governed by and interpreted in accordance with the laws of the Commonwealth of Massachusetts, excluding its choice of law rules.

9. U.S. GOVERNMENT PURCHASES: If SOFTWARE is acquired for or on behalf of the U.S. Government, then:

a. It is recognized and agreed that the SOFTWARE: (i) was developed at private expense; (ii) was not required to be originated or developed under a Government contract; (iii) was not generated as a necessary part of performing a Government contract; and (iv) was not accomplished during and necessary for the performance of a Government contract.

b. It is recognized and agreed that SOFTWARE is commercial computer software, as that term is used in FAR Parts 12 and 27.4 (and any applicable agency supplements thereto), and DFARS Parts 211.70, 227.4 (OCT 1988), 227.71 (JUN 1995) and 227.72 (JUN 1995).

c. If this purchase is subject to FAR Part 27, and FAR 52.227-19 has been incorporated into the terms of this purchase, the Government's rights in the SOFTWARE are no greater than RESTRICTED RIGHTS as specified in FAR 52.227-19.

d. If this purchase is subject to DFARS Part 227 (OCT 1988), the Government's rights in the SOFTWARE are no greater than RESTRICTED RIGHTS as specified in DFARS 252.227-7013(c)(1)(i).

e. The Government's rights in the SOFTWARE are no greater than the rights expressly stated in the body of this Standard Licensing Agreement, if this purchase is subject to (i) FAR Part 12; (ii) FAR Part 27, and FAR 52.227-19 has not been incorporated into the terms of this purchase; (iii) DFARS Part 211.70; (iv) DFARS Parts 227.71 and 227.72 (JUN 1995); or (v) any other regulation or clause permitting the contractor to deliver commercial computer software under the contractor's standard commercial license.

f. Any related technical data shall be delivered to the Government with no more than limited rights.

10. DISCLAIMER OF SOFTWARE WARRANTY: BOSTON DYNAMICS PROVIDES THE SOFTWARE TO LICENSEE “AS IS” AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR WITH RESPECT TO INTELLECTUAL PROPERTY OWNERSHIP, NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY ANY BOSTON DYNAMICS EMPLOYEE, REPRESENTATIVE OR DISTRIBUTOR WILL CREATE A WARRANTY FOR THE SOFTWARE, AND LICENSEE MAY NOT RELY ON ANY SUCH INFORMATION OR ADVICE.

11. Limited Warranty on Media: Boston Dynamics warrants the media on which SOFTWARE is recorded and provided to LICENSEE under this Agreement to be free from defects in materials and workmanship under normal use for a period of ninety (90) days after the date of the original delivery of SOFTWARE to LICENSEE. Such warranty is solely for LICENSEE's benefit and LICENSEE has no authority to assign, pass through or transfer this warranty to any other person or entity. If LICENSEE returns any defective media to Boston Dynamics or an authorized Boston Dynamics representative during the warranty period with proof of purchase, Boston Dynamics will, at its sole option, either replace such defective media or refund the purchase price for such media. This warranty will not apply to any media that has been damaged by abuse, accident or misuse. The foregoing warranty sets forth Boston Dynamics' entire liability and LICENSEE's exclusive remedy for any defects in any media and is in lieu of, and Boston Dynamics disclaims, all other warranties, express, implied, or otherwise, including without limitation any warranty of merchantability or fitness for a particular purpose.

12. LIMITATION OF LIABILITY: IN NO EVENT SHALL BOSTON DYNAMICS OR ITS LICENSORS BE LIABLE TO LICENSEE FOR ANY SPECIAL, CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES OF ANY KIND (INCLUDING WITHOUT LIMITATION THE COST OF COVER, DAMAGES ARISING FROM LOSS OF DATA, USE, PROFITS OR GOODWILL, OR PROPERTY DAMAGE), WHETHER OR NOT BOSTON DYNAMICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSS, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY ARISING OUT OF THIS AGREEMENT. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY. BOSTON DYNAMICS' LIABILITY ARISING OUT OF THIS SOFTWARE LICENSE AGREEMENT AND/OR LICENSEE'S USE OR POSSESSION OF THE SOFTWARE, INCLUDING WITHOUT LIMITATION ANY AND ALL CLAIMS COMBINED, WILL NOT EXCEED THE AMOUNT OF THE LICENSE FEE LICENSEE PAID FOR THE SOFTWARE PROVIDED UNDER THIS AGREEMENT.

13. Laws Governing Warranties and Liability: The law(s) of a jurisdiction may define the scope of warranty to be provided for products or the manner in which a supplier's liability may be limited, and such law(s) shall govern this Agreement only to the extent a party protected by such law(s) cannot waive the protection thereof by contract. In the U.S. and other countries, some states, territories or other principalities do not allow the limitation or exclusion of liability for incidental or consequential damages, or allow the exclusion of implied warranties, so the limitation and exclusion above may not apply to LICENSEE, and LICENSEE may have other rights that vary from state, territory or principality to state, territory or principality.

DI-Guy civilians and soldiers

DI-Guy User Guide - Table of Contents

Introduction

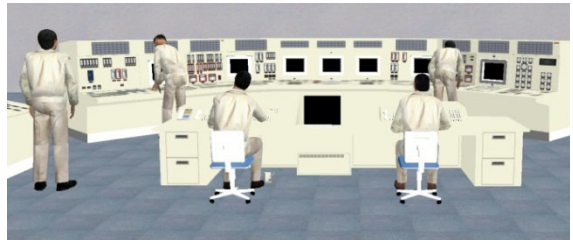
Welcome to *DI-Guy SDK*™!

DI-Guy SDK is software for adding realistic human characters to real-time visual simulations. *DI-Guy* characters move realistically, respond to simple high-level commands, and travel about the environment as directed. *DI-Guy* characters make seamless transitions from one activity to the next, moving naturally like real people.

How is DI-Guy used?

Here are some of the ways people are using *DI-Guy* today:

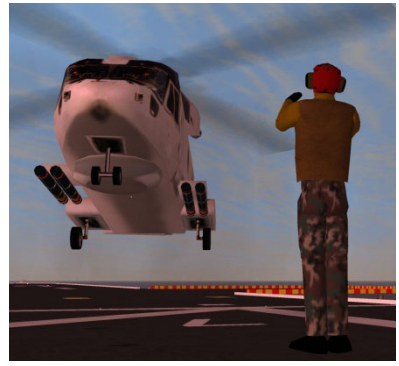
- Ground and urban combat training
- Mission planning and after-action review
- Peacekeeper training
- Law-enforcement training
- Driving simulators
- Urban visualization
- Architectural walk-throughs
- Disaster evacuation planning



DI-Guy is used by all branches of the US Armed Forces and by commercial and military organizations worldwide.

DI-Guy comes with everything you need to add lifelike humans to your simulation:

- Photo-realistic human models that are fully textured, articulated, and come with multiple Levels of Detail (LODs)
- A library of human behavior with over 2000 motions, transitions and gestures
- A high-performance motion engine that lets you command *DI-Guy* in real time
- A powerful, easy-to-use C++ API, the *DI-Guy API.*



DI-Guy includes models and behavior for hundreds of human characters that represent people from all walks of life, including:

- Soldier characters with an extensive array of uniforms, weapons, and behaviors
- Flight deck crew (LSE, LSO, etc.)

- Policemen, firemen, first responders equipped with gas masks and other hazardous (MOPP) gear
- Suspects and enemy combatants
- A wide range of civilian men, women, and children from a variety of cultural and ethnic groups
- Divers, dancers, street sweepers, people using wheelchairs, bicyclists, travelers
- Animals including horses, mules, dogs, cows, kangaroos,…even riders on horseback

DI-Guy also includes hundreds of vehicles and props to support your human performances.

DI-Guy uses industry standard OpenFlight and Collada models and is easy to customize.

High-performance real-time motion engine

An advanced motion engine ensures that *DI-Guy's* behavior is lifelike and realistic. *DI-Guy* comes with over 2,000 motions and transitions. You can modify any of these motions or create and add your own new motions using the optional *DI-Guy Motion Editor*. The *DI-Guy Motion Editor* will audition and export your new motions directly into your characters' action tables for immediate use in *DI-Guy*.

DI-Guy is real-time. *DI-Guy* provides excellent real-time performance through use of optimizations, such as:

- Level-of-detail (LOD) switching
- Motion level-of-detail (MLOD) switching
- Character Performance Level (CPL) switching
- Object-level culling
- The *DI-Guy Load Manager*

DI-Guy works in any Rendering Environment

DI-Guy SDK works out of the box with OpenGL, DirectX, OpenSceneGraph(OSG), and Vega Prime. *DI-Guy* also works with a number of third-party rendering products including:

- Raydon BARE
- Rockwell Collins EPX
- MAK VR-Vantage
- Quantum3D Mantis
- FlightSafety VITAL X
- Lockheed Martin SEView
- CATI X-IG
- DiamondVisionics

If you want to use *DI-Guy* in a simulation environment that is not on this list, the DI-Guy Graphics API module is also available to allow you to integrate tightly with whatever rendering environment you use. Give us a call as we're integrating with new simulation environments all the time.

Networking with DI-Guy

DI-Guy supports DIS, HLA1.3 and HLA1516 networking. The SDK includes functions that simplify the task of getting good human behavior when you integrate DI-Guy into your networked application. We also promote use of the DI-Guy Custom FOM and DI-Guy Custom PDUs to get higher fidelity performances in a distributed networking environment than is available standard with the HLA and DIS protocols. Programming examples are included that show how to integrate DI-Guy into an application that uses MaK's VR-Link product. Note that VR-Link is not sold with DI-Guy SDK, and that you can use the example to help create a networking solution using VR-Link or an alternative networking solution.

Does DI-Guy work with ModSAF/OneSAF/JSAF?

Yes. You can visualize human entities simulated by OneSAF, DISAF, JSAF, ExportSAF and other related SAFS with IGs, Stealths, and viewers enabled with DI-Guy.

The DI-Guy Product Line

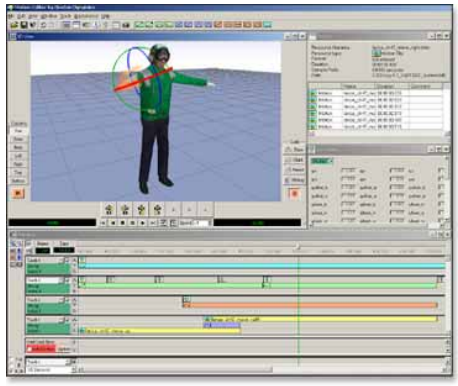
DI-Guy SDK is part of the DI-Guy Product Line, an integrated set of human simulation products. Members of the DI-Guy Product Line are designed to work together seamlessly:

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.



Networking Module – broadcast and receive DI-Guy characters using DIS or HLA. (DI-Guy Scenario only – note that DI-Guy SDK is also routinely networked, see the DI-Guy SDK User Guide on networking to find out how)

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

For more information about the DI-Guy Product Line, go to www.diguy.com or email us at diguy@diguy.com or sales@diguy.com.

Documentation

DI-Guy comes with lots of documentation on how to use DI-Guy:

- ***DI-Guy SDK User Guide*** – How to program using the DI-Guy SDK
- ***DI-Guy SDK Reference Manual*** – All the functions and classes of DI-Guy SDK

- ***DI-Guy Graphics API Reference*** – details about classes and functions of the DI-Guy Graphics API
- ***DI-Guy Scenario User Guide*** – How to use DI-Guy Scenario
- ***DI-Guy Content Guide*** – Images of DI-Guy Content and Customization Guide
- ***DI-Guy AI User Guide*** – How to use DI-Guy AI
- ***DI-Guy Motion Editor User Guide*** – How to import, edit, and customize DI-Guy motions
- ***DI-Guy for Vega Prime*** – How to use DI-Guy with Vega Prime
- ***DI-Guy Digest*** – XML-based file that explains DI-Guy content to non-DI-Guy applications

Once you have installed *DI-Guy* on your system, these documents are available from the start menu under *DI-Guy*, or by browsing to $DIGUY/doc/, where $DIGUY is the location where *DI-Guy* is installed on your system. . 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*.

For more information about the *DI-Guy Product Line*, go to **www.diguy.com** or email us at **diguy@diguy.com** or **sales@diguy.com**.

In summary

Whether you are a seasoned visualization professional or a fledgling start-up, the *DI-Guy SDK* will simplify and speed your task of adding realistic lifelike humans to real-time 3D simulators, letting you save time and money.

- *DI-Guy* automatically creates natural-looking smooth behavior, even when a character changes its behavior from one action to the next.
- You can add characters to a simulation with as few as six high-level function calls, or tweak their behavior and appearance using all 400 function calls available in the DI-Guy API.
- High-level control allows you to concentrate on your application, rather than on the time-consuming details of low-level animation.
- Distributed with easy-to-follow example programs, it takes most users just 1 day to get *DI-Guy* up and running in their application.

This document assumes that you have installed and licensed DI-Guy on your system. See the appendices of this manual for instructions on the Installation and Licensing of DI-Guy.

1. How to use DI-Guy SDK

You can use *DI-Guy SDK* in several different ways, depending on the application and the facilities available to you. In all 3 use cases we discuss below, DI-Guy is used for the visual simulation of the characters. The question here is how the character behavior specified.

Use Case 1 – Author a scenario using *DI-Guy Scenario/DI-Guy AI* and then load the scenario into your *DI-Guy*-enabled simulation and play it. Here the high-level construction is performed off-line in DI-Guy Scenario/DI-Guy AI, a tool designed to quickly and efficiently generate compelling human behavior. RECOMMENDED.

Use Case 2 – Create and control *DI-Guy* characters directly in your simulation via low-level calls to the DI-Guy action API. Here you are doing all the high-level construction locally.

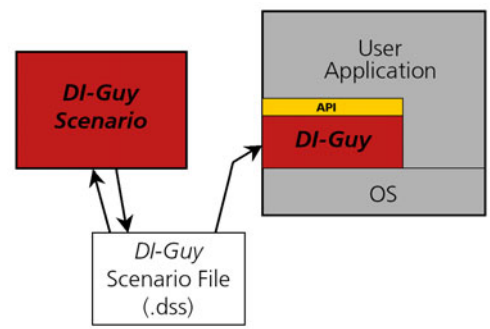
Use Case 3 – Drive *DI-Guy* characters from a network. Examples include being at the end of a DIS/HLA pipe or being in a Host-IG environment. Here the high-level construction is performed somewhere else and piped in.

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. Once you have installed *DI-Guy* you can find the programming examples at $**DI-Guy**/programming_examples/. In addition to OpenGL, examples are provided for DirectX, OpenGL, and OpenSceneGraph (OSG) in the the DI-Guy Graphics API, featuring open source implementatioin 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.

Use 1—Embed scenarios saved from DI-Guy Scenario/DI-Guy AI

DI-Guy SDK is able to 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 totally accessible via the *DI-Guy* API, so you can both monitor and alter their behavior at runtime.



We think this is the most productive way to work with *DI-Guy, because DI-Guy Scenario/DI-Guy AI is designed to let you quickly and efficiently create compelling human performances.* Don't underestimate the bottleneck you may hit if you don't use these productivity-enhancing authoring tools to do your work. We recommend this paradigm whether you are an expert programmer or a non-programming Subject Matter Expert. Either way

you're going to benefit from being able to place and manipulate characters directly in the environment, and then instrument their behavior through an intuitive GUI.

1. Use *DI-Guy Scenario/DI-Guy AI* to author a scenario that contains people, vehicles, props and their interactions.
2. Load the scenario into your application through the *DI-Guy* API.
3. Play.

This method of adding humans is shown in the code example named `simple_playback`.
(Details about programming examples can be found in Appendix E: DI-Guy Programming Examples.)

The following *DI-Guy* OpenGL example loads and then plays back a scenario file created with DI-Guy Scenario running on a Windows computer.

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

```
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. Note that in scene graph style renderers, this step is not necessary.

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

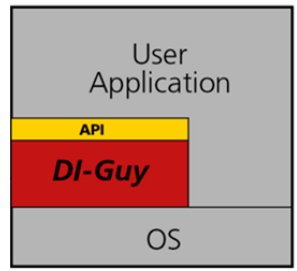
6. When finished, cleanup the scenario and *DI-Guy*.

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

You can find this example in the *DI-Guy* directory:
`%DIGUY%\programming_examples\diguy_ogl\simple_playback` on Windows,
`$DIGUY/programming_examples/diguy_ogl/simple_playback` on Linux.

Use 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 via the feature-rich *DI-Guy* API.



The programming example "simple", described below, creates a soldier character and then commands the character to perform a sequence of movements starting from a standing position.

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*. Scenarios contain characters, events, etc.

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

3. Create a *DI-Guy* character of type *soldier*. You'll need to choose the name, character type, and appearance of your character.

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

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

5a. Before calling scenario_update() in the update loop, make any calls you wish to alter the behavior of your characters. 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 crouched fashion, at 14 seconds he gets on the ground and aims the weapon, and at 18 seconds he

```
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);
```

fires the gun in 1 second intervals. All transitions are handled automatically by the *DI-Guy* motion engine.

6. In your draw loop, draw your *DI-Guy* characters. Note that in scene graph style renderers, this step is not necessary.

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

7. When finished, cleanup the character, the scenario, and *DI-Guy*.

```
scenario->destroy_character(character1);

diguy_destroy_scenario(scenario);

diguy_deinitialize();
```

You can find the source code for this example in the *DI-Guy* directory `%DIGUY%\programming_examples\diguy_ogl\simple`, or run the compiled example by double-clicking `simple.exe`.

On Linux the directory is `$DIGUY/programming_examples/diguy_ogl/simple`., the program `simple`, without the .exe.

Use 3—Network with other simulators (DIS/HLA1.3/HLA1516)

A third common mode of operation for *DI-Guy SDK* is to visualize human characters from the end of a pipe where the actual program controlling the characters is non-local. A typical case is when you connect your simulation to a DIS or HLA network. Another common case is where your simulator is a HOST-IG style simulator, often connected via 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 Lifeforms functions, *DI-Guy* provides custom PDUs for DIS and the DI-Guy FOM for HLA that extend these protocols to fully take advantage of *DI-Guy*'s capabilities.

The following example shows how to network your DI-Guy characters to a DIS network.

It can be found at: `%DIGUY%\programming_examples\diguy_networking\DIS`.

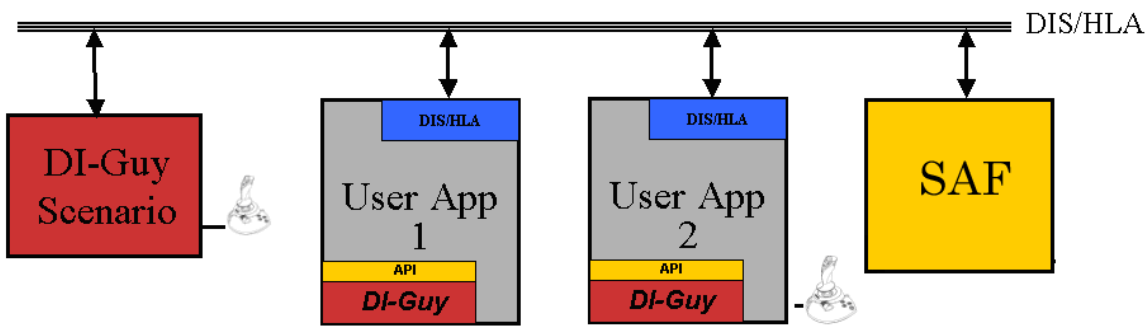


Diagram of a networked environment in which there is interoperation of two applications with DI-Guy SDK embedded, DI-Guy Scenario running as a lifeform server, and a standard SAF. Each system in such a network can broadcast and/or receive DIS/HLA traffic for humans. Systems using DI-Guy will benefit from a higher quality performance.

This example uses MAK Technology's 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 to DI-Guy, you can do so and be successful. We have customers who use VR-Link and others who use their own. Both groups are successful.

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 shutdown, 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();
```

To network DI-Guy using HLA, DI-Guy has also included programming examples for both HLA 1.3 and HLA 1516 at the following location:

- For HLA 1.3 - %DIGUY%\programming_examples\diguy_networking\HLA
- For HLA 1516 - %DIGUY%\programming_examples\diguy_networking\HLA1516

Best Practices for querying and setting actions in a Pipeline Setting (Host/IG or DIS/HLA)

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

1) 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?

2) 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?

The following table shows the 4 basic ways a DI-Guy character can have its action change on a host:

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	set_desired_action()	Smooth, nice transitions, but action will start after current action loop is complete for maximum visual quality.
PATH or FREE	force_action(<action name>, DIGUY_DEFAULT_FLOAT, 1, 0.5f, …)	Smooth transition that starts immediately. Quality of transition is not as nice as set_desired_action() but typically is quite good.
PATH or FREE	force_action(<action name>, DIGUY_DEFAULT_FLOAT, 0, 0.0f, …)	Instant transition with discontinuity.

Host Side

The recommended way to handle the host side 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 doesn't equal the previous_action:

2a) if the current_action does not equal the desired action, send a new action packet with a smooth bit set to 1.

2b) else send a new action packet with a smooth bit set to 0.

There are two drawbacks with this method:

1) The two force action styles are indistinguishable.

2) The set_desired_action() style triggers earlier than it should.

In DI-Guy 10 and higher, these problems are eliminated while also eliminating the need for the user to create the code block above. We now recommend use of the new character callback function diguyCharacter::`CALLBACK_ID_CURRENT_ACTION_CHANGED`. This function will be called when a new action should be commanded down the pipe.

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).

Note that 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, available in the DI-Guy opengl programming examples directory, shows how the guide can be used to best effect.

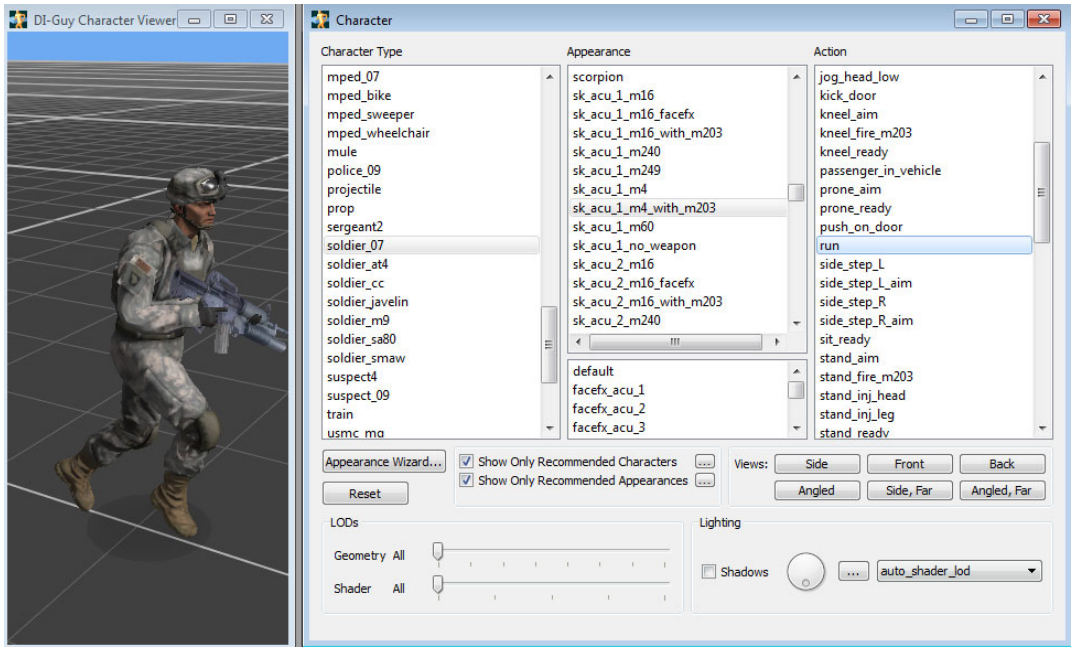
2. How DI-Guy Works

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

The Character Viewer

DI-Guy ships with a license-free application called the DI-Guy Character Viewer. It lets users audition all the characters and motions shipped with each version of DI-Guy. To start the Character Viewer, select it from the Start Menu.



In this image, the DI-Guy Character Viewer has Character Type "soldier_07" slected, with appearance "sk_acu_1_m4_with_m203" and is performing the "run" action.

Assets in a DI-Guy distribution

Directory	Contents
3rdparty	Libraries from other vendors that DI-Guy uses. Included is software for multithreading, licensing, glew, networking, sound, physics, and other utility software.
bin	"Binary" files directory, this contains DI-Guy executables and dynamic libraries.
config	Configuration data describes which character types are available, which appearances are available, etc. and coordinates the geometry and motion assets of the SDK.
custom	Contents in the custom folder will override certain other files in the original DI-Guy installation. This can include config files, geometry, textures, and others.
doc	Documentation.
flexlm	Files needed for DI-Guy licensing using FlexLM license software.
geometry	3D models, textures, and other data needed to render DI-Guy characters including OpenFlight and Collada geometry, png and rgb textures, and compressed and optimized geometry. Reference shaders are located here as well. See following table.
images	Various logos used in documentation and screenshots of most appearances.
include	C++ header files for the DI-Guy API.
lib	C++ library files for DI-Guy-supplied libraries.
me	Runtime data for the Motion Editor.
motions	Motions files for animating DI-Guy characters.
programming_examples	Extensive programming examples showing how DI-Guy can be integrated into various graphics environments and how to use various DI-Guy features.
scenarios	Example scenarios created in DI-Guy Scenario.
sfx	Sounds

Geometry directory contents.

Directory	Contents
compressed_texture_cache	Textures that have been compressed for better load times and smaller memory footprint.
facefx	Files needed for the Expressive Faces module
flt	Geometry files in the OpenFlight format.
fonts	Files used in DI-Guy Scenario for rendering text in OpenGL.
optimized_geometry_cache	Geometry files that are in a DI-Guy graphics pipeline optimized format.
png	Texture files in the .png format.
rgb	Texture files in the .rgb and .rgba formats.
sets	Terrains that can be used in DI-Guy Scenario.
shaders	Reference geoemtry shader programs.

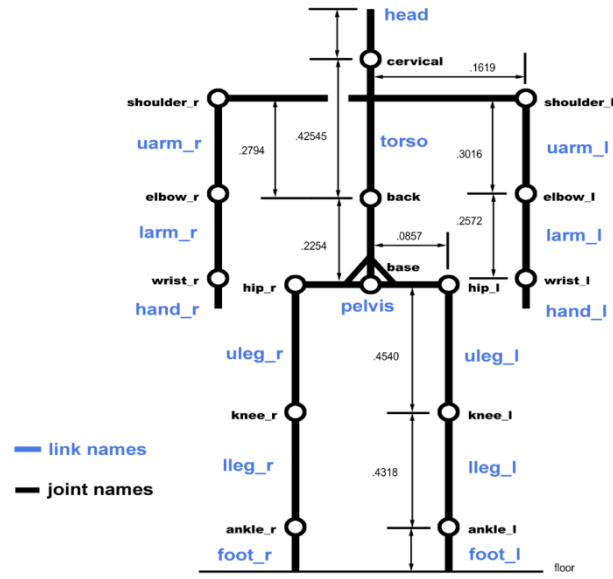
DI-Guy Conventions

Note the following coordinate system conventions used within *DI-Guy*:

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

Diagram of representative skeletal structure for DI-Guy characters, including names of joints and links.

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.



The Basic DI-Guy Function Calls

The following function calls are common to virtually 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()`

NOTE: 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 NETWORKING—`diguy_module_net_initialize()` / `diguy_module_net_deinitialize()`
- HLA NETWORKING—`diguy_module_net_initialize()` / `diguy_module_net_deinitialize()`

(NOTE: DIS vs. HLA networking will be determined by which libraries are chosen to link against.)

- EXPRESSIVE FACES—`diguy_facefx_initialize()` / `diguy_facefx_deinitialize()`
- OPENAL SOUND—`diguy_sound_openal_initialize()` / `diguy_sound_openal_deinitialize()`

Scenario Creation, Population, Update, and Drawing Functions

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 the function `diguy_create_scenario()` to do this.

Once the scenario is created, you need to populate it. Scenarios can be populated 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 via 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()`.

To improve runtime performance, it is sometimes advantageous to update your characters using `diguyCharacter::update()`instead, particularly if you want to skip updating some characters that may not be visible or important at the moment. This performance improving concept is used by the Load Manager.

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.

A Brief DI-Guy Class Overview

The following is a brief description of the two most important *DI-Guy* classes.

> Note that the DI-Guy SDK Reference Manual, along with the Lua AI Minds SDK, should be a users' first and best place to inform themselves of the functions and classes of DI-Guy SDK.

`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 generally 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 are also important *DI-Guy* classes.

`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. Note that the links themselves do not have any geometry directly associated with them; instead 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. Note that as of *DI-Guy*

10, the new 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 pathShape, 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 its travels as you directly specify which action the character is performing. The combination *free position/path action* mode is not allowed.

`diguySceneObject`
A scene object is a static (i.e. – non-moving) objects in the scenario to which characters can sometimes be ground clamped. Scene objects are typically sections of terrain, such as a small town. Scene objects do not typically animate.

`diguyView`
A view is a graphics window where the scenario and its characters are animated. Note that 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.

Best Practices

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 destruct *DI-Guy* classes using new and delete functions – in fact, these constructors are private and cannot be called. Often these classes will automatically get destructed when the higher class is destructed. 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[1]. 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*. The following table shows how the two modes interact:

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

[1]DI-Guy characters can also be controlled via methods described in the DI-Guy AI manual. 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.

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 is an overview of some of the key functions and concepts of each class.

diguyScenario

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. Note that no rendering happens in `update()`.

On immediate mode rendering systems, rendering is performed in the `draw()` function. The draw function actually calls two sub-functions, `draw_pass1()`, which draws the opaque objects in the scene, and `draw_pass2()`, which draws the transparent objects. Users needing that level of drawing can use these functions.

Functions are provided for turning on and off the culling of characters that are outside the view frustum. Characters are surrounded by bounding spheres.

Working with .dss files

You can load a diguyScenario class from .dss files created by *DI-Guy Scenario* using the `load()` command.

Time Control and Playback Functions

The `reset()` function rewinds the scenario back to the start time (*tin)* 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 `set/get_tout()`, `get/get_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 *tin* is available in *DI-Guy* and can be accessed through the `tin_time_of_day` family of functions.

Improving Performance Functions

There are functions available for improving *DI-Guy* loading performance. `diguyScenario::preload_character_type(), diguyScenario::preload_gesture()` and `diguyScenario::preload_appearance()` allow the user to avoid hiccups at runtime which would occur if new character data were loaded when a character or gesture were called for the first time. These functions load the data during startup when delay is less important.

There are a number of features available for improving *DI-Guy* updating performance. These include a multi-threaded control API in diguyApp, see `diguyApp:set_num_threads()`. The `get/set_ticks_can_be_dropped()` and `get/set_max_ticks_behind_till_dropped()` allow the user to tune scenarios that have trouble running in realtime. Users interested in performance functions should read the later chapter on Maximizing Performance of DI-Guy.

General Query Functions

There are a number of functions available in the diguyScenario class that can be used to query about general *DI-Guy* content capabilities. You can query for:

- 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. Use `get_num_characters()` to find out how many characters are in the scenario. Functions are then available to retrieve these characters either by name or by index. Use `create_character()` and `destroy_character()` to instantiate and destruct characters.

We call a *DI-Guy* character that is controlled by a joystick an *I-Guy*. Use `set_iguy_character()` to designate a character as being driven by a joystick.

`add_custom_appearance()` and `add_custom_appearance_based_on_existing_appearance()` allow users to construct new custom appearances for characters.

Path Shape and Waypoint Functions

Path Shapes are the splines that underlie paths in *DI-Guy*. The diguyScenario class offers a number of 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, as well as retrieve the number of signals, retrieve a signal via its index, or retrieve a signal via its name. All of the signals can be reset using `reset_signals()`.

Sound Functions

You can create and destroy sounds, as well as retrieve the number of sounds, retrieve a sound via its index, or retrieve a sound via its name. All sounds can be reset using `stop_all_sounds()`
Note that sounds are not played directly. Instead they are used as templates for sound instances. 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. When networking for example, combatants will be classified as belonging to groups

net_friendly, net_opposing or net_neutral. You can create and destroy groups, as well as retrieve the number of groups, retrieve a group via its index, or retrieve a group via 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 via its index. Sensor regions can have callbacks associated with them for when characters enter or leave.

Scene Object Functions

Scene Objects are stationary objects (usually terrains) to which other waypoints can be ground clamped. You can create and destroy scene objects, as well as retrieve the number of scene objects, retrieve a scene object via its index, or retrieve a scene object via its name. Scene objects can be enabled or disabled (i.e. 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()` will set the primary view's camera parameters from the specified camera setting.

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

Fog and Light Functions

There are a set of functions for managing both 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 for users wishing to program while running inside of *DI-Guy Scenario.*

Info Popup Functions

Info Popups are boxes with HTML text that appear when running a scenario inside of the *DI-Guy Scenario* application. 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.

Face Expression Functions

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

Callback Functions

Users can use the `add_callback()` function to have a function called when any of the scenario-level events listed below occur. Note that some of the events are *DI-Guy Scenari*o application specific.

- `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

Default Character Callbacks, Altitude Functions, and other Default Callbacks

The function `add_default_character_callback()` allows users to assign particular character level callbacks to all subsequent characters added to the scenario. Similarly, `set_default_character_altitude_function()` allows users to assign a particular altitude function to any character subsequently added to the scenario. Similarly, there are methods for assigning callbacks to paths, sensor regions, signals, views, cameras, fog, and lights that are subsequently created or added to the scenario.

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 a wealth of functions for registering various event handlers tied to signals, sensor regions, character events, and scenario events.

History/Review Functions

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

LOD Functions

DI-Guy supports Level-of-Detail for both *Graphics LODS* (where characters are drawn with various numbers of polygons in a detail vs. performance trade-off) and *Motion LODS* (where characters can limit which joints are updated in a detail vs. performance trade-off). 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 a user 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, a user might ask for a moving forward motion that goes 4 MPH in an upright posture. For a soldier, the “jog” action might be returned, while for the female_pedestrian, perhaps “powerwalk” would be returned. Note that variant can mean different things, one example might be that “funky” vs. “stiff” might be a way of differentiating between two different types of walking.

diguyCharacter

General Accessor Functions

Use `get/set_name()` to access the name of the character. Its character type can be accessed using `get/set_type_name()`. You can retrieve the scenario the character belongs to using `get_scenario()`, and determine its enabled status with `get/set_enabled()`. You can use `set/get_drawn_by_scenario_flag()` to determine whether the character gets drawn when its parent scenario gets drawn.

Use `set/get_current_tin()`, `set/get_current_tout()` to set the time range during which the character will be active.

Use `set/get_position()` and `set/get_orientation()` to set the geographic position/orientation of the character. Note: `get/set_position_double()` is available as well for when the character needs to be far away from the origin. Use `set_position_relative_to_parent()` and `set_orientation_relative_to_parent()` when parenting the character to another character. Use `set_initial_position()` and `set_initial_orientation()` to set the position of a character when a reset occurs. Note 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.

Use `get/set_scale()` to access the scale of the character along the 3 major axes.

`is_group_member` () returns whether the character belongs to a particular group.

DI−Guy Character Update and Draw Functions

U`pdate()` advances time forward to time *t* and new position and joint values are calculated for the characters. Note that no rendering happens in `update()`. Use `set/get_t_controlled_by_scenario_t()` to check 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 actually calls two sub-functions, `draw_pass1()`, which draws the opague objects in the scene, and `draw_pass2()`, which draws the transparent objects. Users needing that level of drawing can use these functions.

Appearance Functions

Access the character's appearance using `get/set_appearance()` and `get/set_current_appearance()` Note that the 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. Note that this means if the scenario is reset, the appearance will revert to base appearance.

A Note on Changing Character Appearances when Appearances Reference Different Actors - 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.

`Set/get_current_head_appearance()` allows you to swap heads on the character.

Dying

`die_now()` is called to tell a character to die as soon as possible. Characters will execute their dead action. All aiming, gazing, pointing, head nodding, gestures, and sounds will be stopped. Use `get_dead()` to retrieve the dead state of a character. Call `revive_now()` to resuscitate a character.

Improving Performance Functions

Use `get/set_motion_interpolation_flag()` for accessing whether the motion data of the character is interpolated. This is disabled by default for better performance. For very high frame rate performances, smoother motion may be possible by turning interpolation on.

Character may be culled in or out of the scene. You can set or view the characters bounding radius by using `get/set_bounding_radius()` and `set/get_default_bounding_radius()`. Character viewer can also display them. 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 want to alter the up vector. A good example would be a vehicle like an airplane banking as it made a turn. Use the `set/get_up_vector()` to access this value.

Action/Speed Functions

Use `set_speed()` to set 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.

Use `set_desired_action()` to put the character into Free Action Mode. Note that 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 Postion Mode for Free Position Mode. There is a `get_desired_action()` function as well.

A related function is `force_action()`. It has a similar effect on the various modes as `set_desired_action()`. The difference between the two calls is that `force_action()` forces the new action to be blended to immediately, while `set_desired_action()` allows the current action to complete its cycle before transitioning.

`force_action_duration()` is for cases where the character is in Path Position mode, and you want 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.

Parenting Functions

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

Link and Shape Functions

Use `get_link_position()` to get 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(_double)()` allows you to provide an offset into that link.

The `get_link_relative_position()` returns the relative position and orientation of the link relative to the parent link. There is an also a `with_offset()` version.

Retrieve particular links using the `get_base_link()`, `find_link()`, and `get_link_at_index()` functions.

LOD Functions

Call `set/get_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, etc.)

`set/get_motion_lod()` allows access the motion LOD state of the character. Coordinate use of the set function with the Load Manager.

User Data

`get/set_user_data()` allows users to attach and access their own classes or structures to a *DI-Guy* character.

Path Functions

Use the function `set_distance_along_path()` to position a character at a particular distance along a path when in Path Position Mode. `Leave_path()` will make 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.

There are a host of 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()` will take a character off the current path and put him on the input path. `force_partial_path()` allows you to designate 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 transistions the character onto the new path.

`create_path()` and `create_simple_path()` allow for the 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()` will play an "ambient" (positionless) sound.

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()` will query whether any characters that belong to a particular group are near your character.

Callback Functions

Users can use the `add_callback()` function to have a function called when any of the character-level events listed below occur. Please refer to the SDK Reference Manual for a list of Callbacks. Types of callbacks include upon create, upon destroy, pre and post weapon fire, etc.

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 provides a wealth of 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 either being directly commanded or by reaching an Aim, Gaze, or Bead on the path they are traveling.

Aim, gaze, and point have a wide range of functions for specifying how to aim, gaze, and point, whether it is at a point in space, a local direction, at another character or link. Functions are also available for determining the status, duration of the aim, gaze, or point, and also 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 either setting the position manually (via 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 via some type of graphics intersection testing.

Weapon Functions

Characters in *DI-Guy* with weapons can fire them. There are functions available for determining the muzzle flash and sound to play when firing. Use the function `set_weapon_fires_live_rounds()` to determine whether the character will potentially hit/kill other characters. The kill zone has a near, far, and weapon spread that is programmable.

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

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.

Many gestures proceed in stages. 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.

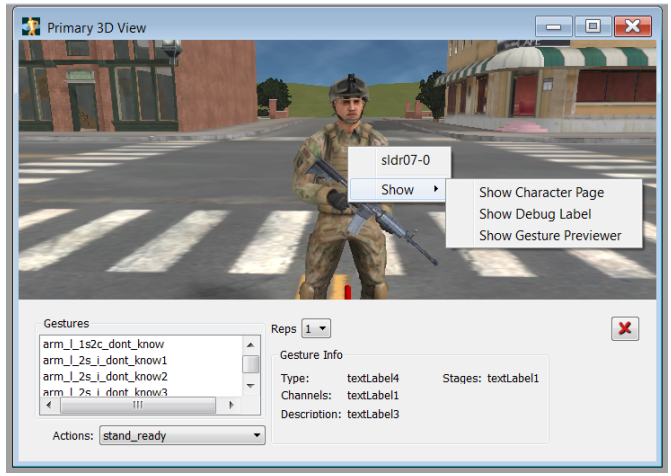
An example of a three stage gesture could be a wave. In stage 1, the gesture raises the character's arm. In stage 2, the arm waves back and forth once. Stage 2 may be repeated multiple times to get the character to wave more than once. Stage 3 lowers the arm back to the character's side.

Note that stages are numbered 1 to 3, not 0 to 2.

Some gestures have more than one channel. Each channel in a multi−channel gesture contains a variation of the gesture. As the gesture progresses, the character may move from one channel to another.

An example of a two channel gesture could be a chop beat gesture for emphasizing speech. Channel A is a small, tentative chop. Channel B is a strong, emphatic chop. If the weights of channels A and B are 1 and 0 respectively, the gesture will be the small chop. If the weights are 0 and 1 respectively, the gesture will be the emphatic chop. If the weights are 0.5 and 0.5, respectively, the gesture will be a combination −− partly small, partly emphatic.

DI-Guy Scenario has an interactive mode that allows gestures to be easily previewed, right click on any character and do show Gesture Previewer.



See the SDK reference for details on the gesture functions available.

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'll 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 wish 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.

See the SDK reference for details on using sound instances.

DIS Functions

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

Formation Functions

A *DI-Guy* character can call or break a formation.

Working with Guides

A guide is something that can influence the position, orientation, or action of a character that is in free position mode and free action mode. Guides are most often used by either 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 will look at the desired position 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 explicitly set by the calls `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 `diguyCharacterGuide::set_guide_algorithm_float_parameter()`.

Once a guide has been added to a character it can potentially 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 four algorithms are offered as Guides:

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

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.

3. The Motion Pipeline

When *DI-Guy* determines the values for all the degrees of freedom of a character during an update, it follows a deliberate sequence as illustrated in the table below:

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. Advanced users may even wish to insert their own modifications to the final pose of the character in Stage 0 or Stage 9.

The Initial Stage

When DI-Guy advances in time, underlying motion files are accessed and data values are retrieved for the degrees-of-freedom of the character. The particular data values are retrieved by indexing into the motion file. Sometimes blending occurs when processing transitions from one looping action to another.

Stepping Gaze is also applied during the initial stage. A stepping gaze occurs when a character needs to look at something beyond his normal gazing range. In this case, the character will actually shuffle his feet to rotate his body so that he is capable of achieving the goal gaze.

Aiming is performed by interpolating in joint space between 2 or 4 aim poses that define the extremes of the aiming space. These values are also calculated during the initial stage.

Gaze and pointing

Use the Gaze and Pointing API to have a character stare or at point at a particular location in space.

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

- At a point referenced locally
- At a point referenced globally
- At a character
- At a link on a character
- Via an azimuth and elevation

Other API functions allow you to access and modify how quickly the gaze or pointing takes effect, as well as 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*).

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

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, including aiming:

- At a point referenced locally
- At a point referenced globally
- At a character
- At a link on a character
- Via an azimuth and elevation

The Aim API let's 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.

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

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.

See the *DI-Guy SDK Reference* for a detailed functional description of User Pose Override capabilities.

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.

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

Head nodding and shaking

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

See the *DI-Guy SDK Reference* for a functional description of these capabilities.

Motion textures

Motion textures are a way of adding some natural looking variability to a looping motion. Motion textures are usually a relatively long motion designed to give realistic randomness to a motion that would otherwise look too repetitive. Sometimes the motion texture is also meant to add an emotive context to the base motion.

See the *DI-Guy SDK Reference* for a functional description of motion textures.

4. Compiling and Linking DI-Guy on Windows and Linux

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.

Compiling and Linking DI-Guy on Windows

Programming Examples

Windows programming example programs and makefiles are in `%DIGUY%\programming_examples`.

The easiest example program is to start with is `%DIGUY%\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.

DI-Guy Library and Header Files

The *DI-Guy SDK* header files are located in the directory `%DIGUY%\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 the 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 the directory `%DIGUY%\lib\win\(compiler_version)\`, where (compiler_version) is, obviously, 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`.

md vs. mdd vs. mdn Versions

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, etc.) The common versions are Multi-threaded Static, selected with the /MT flag at compile time, and Multi-threaded DLL, selected with the /MD flag. Both have debug versions, selected by the /MTd and /MDd compiler flags respectively. 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 the the same program, but this can somtimes 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.

Lua Script Modules

Refer to the `diguy_ogl\simple_playback` example for an example of using the Lua scripting module.

Expressive Faces Module for OpenGL

Refer to the `diguy_ogl\facefx` example for an example of using the Expressive Faces module.

Compiling and Linking DI-Guy on Linux

Programming Examples

Linux programming example programs and makefiles are in `$DIGUY/programming_examples`.

The easiest example program is to start with is `$DIGUY/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 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.

DI-Guy Library and Header Files

The *DI-Guy SDK* header files are located in the directory `$DIGUY/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 the 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 the directory `$DIGUY/lib/linux/(compiler_version)/`, where (compiler_version) is, obviously, 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`.

The *DI-Guy SDK* libraries for Linux were compiled on a Red Hat Enterprise 5 Linux.

5. Techniques for Maximizing DI-Guy Performance

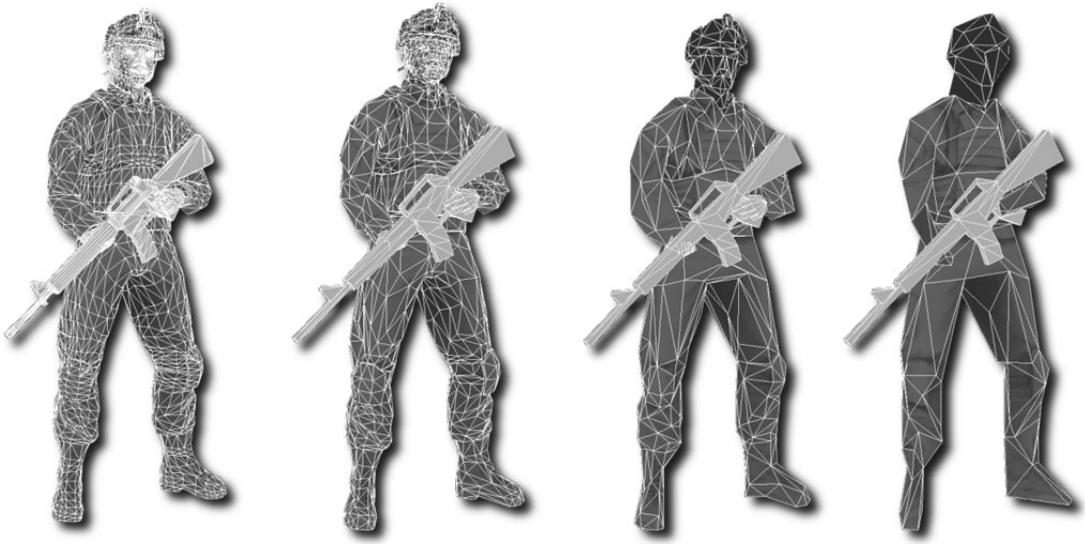
DI-Guy provides a number of tools, techniques, and functions for improving the runtime performance. You can use these tools to create simulations that look great and run fast. In this section *performance* refers to the efficient update and drawing of *DI-Guy* Characters so as to maximize graphics frame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

There are a number of basic techniques that all users should be aware of to scale performance of DI-Guy characters:

1. Draw Culling: If a character is not visible in the view frustum, eliminating a draw call to that character will improve performance.

2. 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 perhaps entered the view frustum). It is often also important to stagger these updates to maintain performance consistency.

3. Graphics LOD: Graphics LOD (Level of Detail) specifies how detailed a model to display. In the models below, we see 4 LODs, with polgon 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.

You can set the graphics LOD of a character using the diguyCharacter function call set_graphics_lod(), however, typically LODs are controlled automatically. Users have the ability

to 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.

4. Multi-Threading: DI-Guy takes advantage of multiple cores to animate it's characters, use diguyApp::set_num_threads() to control how many different threads DI-Guy will use for animating.
5. Motion LOD: Motion LOD (Level of Detail) indicates how many of the joints of a character should be updated to perform its motion. An out of view character may only need to update its position. If a character is far away but in view, you still may be able to just update its position with no visual degradation of the character. Similarly, as you get closer, you may choose to update everything but the wrists and ankles and see no degradation in the visual appearance of the character.
    a. Motion LOD 1: Animate all joints
    b. Motion LOD 2: Same as LOD 1 except no wrists and ankles animation
    c. Motion LOD 3: Same as LOD 2 except no elbows and knees animation
    d. Motion LOD 4: Only animate pelvis and position
    e. Motion LOD 5: Only animate position

    Set the motion LOD of a character using the diguyCharacter class function call set_motion_lod().

6. The DI-Guy Load Manager: The DI-Guy Load Manager allows you to apply the 4 culling and LOD methods described above to a single view application in a simplified manner so that you don't need to get bogged down in the details. The Load Manager is described in the next section.

7. The Character Performance Level: The CPL allows you to balance the advanced capabilities of *DI-Guy* characters with your performance needs. If a character is not using advanced features, you should consider increasing that character's minimum CPL. The CPL levels work as follows:
    a. CPL 1: All character function calls are available.
    b. CPL 2: The characters cannot save history.
    c. CPL 3: no advanced pose operations that alter the basic pose of the character, including:
        i. Gazing
        ii. Pointing
        iii. Aiming
        iv. Gesturing
        v. Head Nodding and Shaking
        vi. Pose Overrides
    d. CPL 4: no blends between motions.

    Use Functions such as set_automatic_cpl_switching_enabled() and set_current_cpl() in the diguyCharacter class to access this functionality in your code.

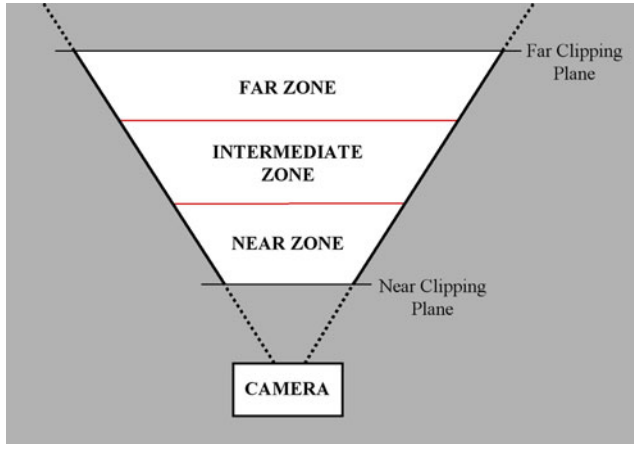
8. Shader LODs: By use of DI-Guy shader techniques, simpler shaders can be instrumented for characters farther away from the origin.

9. Advanced Performance Techniques: There are a number of advanced performance techniques listed below that can be applied, sometimes to dramatic effect, to increase your *DI-Guy* performance. The use of skinned vs. segmented characters often affects the relevance of these techniques. A Benchmark program is included to help measure the effect of these advanced performance options.

    a. Disabling Motion Blends
    b. Disabling Update
    c. Disabling Draw
    d. Disabling Motion Accumulation
    e. Disabling Shape Callbacks
    f. Enabling Individual Character Updates
    g. Enabling Optimized Updates
    h. Enabling Cached Pointers
    i. Enabling Quaternion Precomputation
    j. Enabling Axis Motions Optimization (Aim)

Using the Load Manager

When there are many characters in a scene, the *DI-Guy Load Manager* (class diguyLoadManager) 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, and some character joints are removed from the articulation process (such as wrists and ankles) to further reduce computing load. 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. 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. Please note that the Load Manager does not support multiple viewports.

Enabling Load Management – By default, the Load Manager is disabled. To enable the Load Manager, simply 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 to assign 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
```

Out-of-Frustum – setting Maximum Update Period – 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.

Performance Use Case: DI-Guy as part of a networked Image Generator

There are a myriad of ways to use *DI-Guy*, and thus a myriad of combinations in which to apply the *DI-Guy* performance improvements functions discussed above. A typical use, which we will illustrate here, 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 via DIS or HLA.

Draw Culling, Update Culling, Motion LODS, and Load Management – As discussed earlier, 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 *DI-Guy Load Manager*. You'll need 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.

Character Performance Level – set all your character performance levels to '3' since you will not be using advanced pose operations such as aiming. Note that characters will still react to "aim mode" information from the network by performing actions such as "kneel_aim", but that azimuth/elevation-style modifications to the upper body and weapon vector are disabled (i.e. – the character always aims forwards). Do not use CPL 3 if you are aiming with vehicle_turret or vehicle_technical characters.

Disable Motion Accumulation – do this if you are setting the position of your *DI-Guy* character every frame. If you are only periodically updating, and then correcting position using 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, see the diguyGraphicsShaderTechnique class for more details.

Disable Graphics API Shape Updates – if you're not sure if you are using *DI-Guy* Graphics API Shape Updates, then you probably aren't, so disable them.

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. Note that disk 3 of DI-Guy data includes the uncompressed version of these models. By default, when loading a model that does not have compressed geometry, DI-Guy will attempt to compress and save the .bdg file for future use.

DI-Guy 12 ships with compressed textures. Since DI-Guy 12 supports multiple textures, it is vital that users compress using our recommended DDS texture format to maintain high performance. The executable DIGuyGeometryProcessor will take source .png and .rgb format textures and output .dds files using NVIDIA texture tools and lossless compression (don't worry, DI-Guy will do this automatically for you).

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 config file during initialization, it merely 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.

DI-Guy functions that load geometry and texture for an appearance:

- `diguy_*_initialize()` loads appearances with "preload=true". Note that by default DI-Guy always sets characters to have "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.

DI-Guy functions that load motion data for character actions:

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

DI-Guy functions that 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.

Improving Loadtime Performance

DI-Guy is configured by default to give you access to all the characters available. A survey of DI-Guy 12 showed that the default configuration file diguy.cfg referenced:

- 86 character types
- Over 7000 unique character type / appearance combinations
- Over 175 gestures
- Over 3700 motion arcs

That's a lot of data to catalog! If you know beforehand the characters your scenario will be using, you can improve loadtime performance by specifying your own configuration file that culls out elements you don't need. You can specify your new configuration file be used with a call to diguy_set_cfg_file() before calling diguy_*_initialize(). We recommend that you put your new configuration in the custom/config directory.

For a quick start:

1. Copy diguy.cfg from %DIGUY%\config to %DIGUY%\custom\config\myfastdiguy.cfg.

2. If you are not using expressive faces, gestures, or motion textures, use the '#' character to comment out the lines that refer to them.

3. Change the reference to diguy_characters.cfg to be myfastdiguy_characters.cfg. Change the reference to diguy_actors.cfg to be myfastdiguy_actors.cfg. Change the reference to diguy_appearances.cfg to be myfastdiguy_appearances.cfg.

4. Copy diguy_characters.cfg from %DIGUY%\config\diguy to %DIGUY%\custom\config\diguy\myfastdiguy_characters.cfg.

5. Delete or comment lines referring to characters that you will not be using.

6. Copy diguy_actors.cfg from %DIGUY%\config\diguy to %DIGUY%\custom\config\diguy\myfastdiguy_actors.cfg.

7. Delete or comment lines referring to actors not needed by your cast of characters.

8. Copy diguy_appearances.cfg from %DIGUY%\config\diguy to %DIGUY%\custom\config\diguy\myfastdiguy_appearances.cfg.

9. Comment our lines referring to unused appearances and change the reference to appearances_base.cfg to be myfastappearances_base.cfg.

10. Copy appearances_base.cfg from %DIGUY%\config\diguy to %DIGUY%\custom\config\diguy\myfastappearances_base.cfg.

11. Comment out any unused appearances.

A Note on 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:

1. include <file>: Replace the line with the contents of file. Report an error if file cannot be read.

2. 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.

3. include_optional <file>: Replace the line with the contents of file. Do not report an error if file cannot be read.

4. 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.

6. Networking with DI-Guy

Since its inception, *DI-Guy* has always been designed to support DIS and HLA networked simulations.

There are *DI-Guy* functions that map the standard entity, fire, and detonation from DIS PDUs and from RPR FOM HLA data into *DI-Guy* function calls. There are also functions that map behavior created using the *DI-Guy API* into calls that produces standard DIS or HLA traffic.

DI-Guy Custom PDUs / Custom FOM for high fidelity visual and motion correlation

In addition, *DI-Guy* offers sophisticated extensions to standard DIS and HLA that enhance the functionality and versatility of human characters in distributed simulation. These extensions are called *DI-Guy Custom PDUs for DIS* and the *DI-Guy FOM Extension for HLA*.

> DI-Guy Custom PDUs/FOM lets users 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 actual DIS and HLA software that talks to the network 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 MaK's *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. (See www.mak.com to purchase VR-Link.) Unlike DI-Guy SDK, The DI-Guy Lifeform Servers and DI-Guy Scenario with the networking option provide full networking capabilities, internally they use VR-Link.

Using DIS or HLA networking, *DI-Guy* is already being used to interoperate with third party applications including *MÄK VR-Forces,OneSAF, DI-SAF, Presagis STAGE and Vega Prime, JSAF, VBS2, etc.*

How do custom PDUs/custom FOM work?

They work by augmenting the standard communication for human entities with extra information. When using DIS, the extra information is conveyed in *DI-Guy* custom PDUs. For HLA, the *DI-Guy* FOM is used to convey this extra functionality.

What if I have a network where only some of the stations use DI-Guy?

No problem. Because this is extra information, all the standard information for moving human entities is still there. Machines that have DI-Guy will have better-looking performances.

Do custom PDUs/custom FOM do anything besides improve the fidelity of my characters' movements and appearance?

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

So if I want to see a wide range of characters on the receiving end with a multitude of appearances performing motions beyond the basic stand/walk/jog/etc. , I should use custom PDUs?

Yes!

In summary:

- If you are satisfied with the current fidelity of your networked humans, you are fully supported by *DI-Guy* Networking with no muss, no fuss.

- If you want to take your networked human simulation to the 'next level', you are encouraged to use custom PDUs/custom FOM to achieve cutting edge networked human simulation.

DIS networking using the SDK

DI-Guy broadcasts and responds to standard DIS 2.0.4 entity state PDUs. It can also broadcast and respond to *DI-Guy* Custom DIS PDUs. *DI-Guy* also broadcasts and responds to standard DIS 2.0.4 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. Finally, after the *DI-Guy* scenario object has been destroyed, call:
   `diguy_module_net_deinitialize()`

See the "*DI-Guy API, VR-Link DIS Module"* chapter of the *DI-Guy SDK Reference Manual* for more information about these functions, as well as a listing of other DIS-related functions offered by *DI-Guy*.

Note: the diguyNetInterface class acts a wrapper around either a DIS or a HLA (HLA1.3/HLA1516) network interface, the type of interface that is created depends on which library is linked against.

DI-Guy supplies example code that demonstrates broadcasting:

`$DIGUY/programming_examples/diguy_networking/DIS/net_send`

and example code for demonstrating receiving DIS see:

`$DIGUY/programming_examples/diguy_networking/DIS/net_recv`

Controlling 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 either:

- the `translation_threshold` (default 0.05m)
- the `rotation_threshold` (default 0.2 radians)
- or if the `timeout` threshold is reached (default 5 seconds). The timeout rate can be alter by calling `diguyNetInterface::set_timeout_interval()`

These threshold parameters are specified in the configuration file `network.cfg`, located in:
`$DIGUY\custom\config\diguy\scenario`

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 the `network.cfg` file. 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 */
}
```

- `m_pdu_header`—the PDU record header described above.

- `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, you should ignore the message.

- `m_bdimessageid`—Is 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,
.
.

};
```

Please note that a second set of message ids, the *DI-Guy Scenario* Control PDUs, are discussed in a later chapter. 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.

`m_bdimessageid`		`m_data` format	`m_data` meaning
1	`SET_CHAR_TYPE`	String	Character type
2	`SET_CHAR_APPEARANCE`	String, String	Character appearance, head appearance
3	`SET_CHAR_ACTION`	String, String	Character Action, Paused Status (see below)
4	`SET_CHAR_VISIBLE`	Long Boolean	Visible/Invisible

For the `SETCHARACTION` PDU, the first string is the name of the action. The second string is either “p” or “” (the empty string). A value of “p” for the second string indicates that the action is paused. Characters are typically paused if the sending application such as DI-Guy Scenario has been playing and the user hits “pause” on the VCR control.

For further information and type definitions, consult IEEE 1278.1.

NOTE: 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.

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, described below, 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.

In the event that neither custom PDUs nor an entity map specify the character and appearance, *DI-Guy* displays/sends the entity as either:

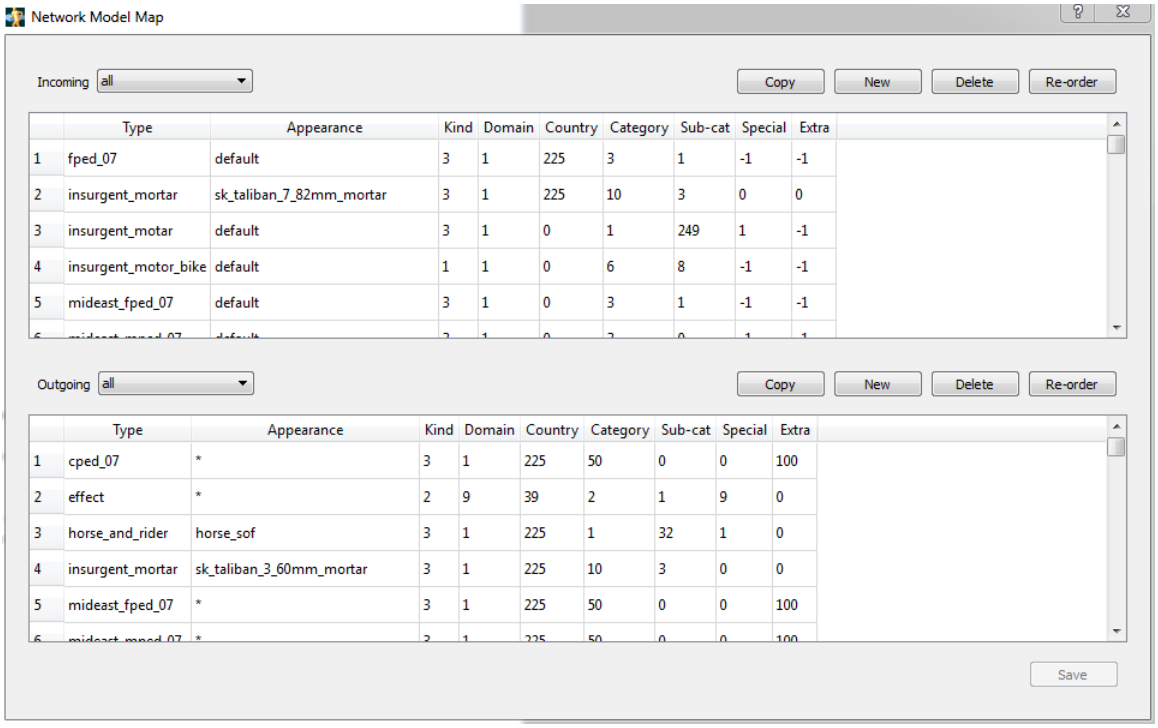
- An American soldier (lifeform)
- An American M1A1 Abrams tank (vehicle)

Specify the desired mappings in the configuration file `network_model_map.cfg`. The installed version is located in the directory:

```
$DIGUY\config\diguy\scenario
```

Note: The network_model_map.cfg file should be placed in $DIGUY\custom\config\diguy\scenario. All custom changes should then be made to this custom version of the file. The custom network_model_map.cfg file will always override the installed version.

DI-Guy offers a network model map application as part of DI-Guy Scenario with an easy-to-use GUI to update/modify this file (RECOMMENDED).



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 solder for lifeforms and an American M1A1 Abrams tank for platforms.
- Comments on the end of each line are ignored.
- The two word pairs (e.g. – "vehicle2 helicopter1" or "soldier2 default") refer to the *DI-Guy* character and appearance.

```
map incoming
entry =  1 -1 -1 -1 -1 -1 -1 = none default            don't map unknown
vehicles
entry =  1  1 -1 -1 -1 -1 -1 = vehicle  Bradley        land vehicles are
bradleys
entry =  1  2 -1 -1 -1 -1 -1 = vehicle  helicopter1    air vehicles are
helicopters
entry =  1  3 -1 -1 -1 -1 -1 = vehicle  boat           surface vehicles are
boats
entry =  3 -1 -1 -1 -1 -1 -1 = soldier07 default        life forms are soldier2
by default
entry =  3  1 78  1 32  1  1 = soldier07 taliban_1
entry =  3  1 78  1 32  1  2 = soldier07 taliban_2
entry =  3  1 78  1 32  1  3 = soldier07 taliban_3
entry =  3  1 255 1 32  1  1 = soldier07 north_alliance_1
entry =  3  1 255 1 32  1  2 = soldier07 north_alliance_2
entry =  3  1 255 1 32  1  3 = soldier07 north_alliance_3
entry =  3  1 225 1 32  1 255 = fped07 default
entry =  3  1 225 1 32  1 254 = mped07 default
entry =  5  2 -1 -1 -1 -1 -1 = symbol   default air cultural features are
symbols

map outgoing
entry = symbol * = 5 2 0 21 0 0 0              symbols are cultural features
entry = vehicle * = 1 1 225 0 0 0 0                 vehicles are tanks
entry = vehicle helicopter1 = 1 2 225 0 0 0 0 helicopters are flying
```

network_model_map.cfg

HLA networking

DI-Guy supports HLA (High Level Architecture) interoperation for HLA1.3 and HLA1516. *DI-Guy* subscribes to and publishes standard RPR-FOM (Real-time Platform Reference - Federated Object Model) attributes, fire and detonation interactions, and *DI-Guy* Custom attributes described below.

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.

RTI compliance

No RTI is delivered with *DI-Guy*. 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 adding *DI-Guy* into the mix.

Setting up DI-Guy to use HLA

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

Step 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\makRti4.0.4\lib
```

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

Step 2: Install and enable 3rdparty networking

DI-Guy SDK does not come with networking software but demonstrates how to use DI-Guy with such software in the programming examples. Remember to purchase and enable 3rdparty networking.

Step 3: Choose an RID file

Most RTI's require a `RID` (RTI Initialization Data) file. Use an `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`. In fact, `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. Some sample `RID` files exist in `$DIGUY/config/diguy`.

Step 4: Choose a federation file

A `FED` file (Federation Execution Details) 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.

DI-Guy recommends a `FED` file that includes the *DI-Guy FOM Extensions*. Note: For HLA 1.3 the standard defines this file using the extension *.FED. For HLA1516 the extension and format has changed to XML.

The *DI-Guy FOM Extensions* are provided in:
`$DIGUY/config/diguy/DI-Guy.fed` for HLA 1.3
`$DIGUY/config/diguy/DI-Guy.xml` for HLA1516

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 `$RTI_CONFIG` is not defined, it assumes a value of:
`$DIGUY/config/diguy`

The `FED` file name used by the RTI is specified by the `federation_name` line in:
`$DIGUY/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.

Enabling HLA networking in DI-Guy

Once you have set up your computer for HLA networking as described above, 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 isconstructed,
   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. Finally, after the scenario object has been destroyed,
   call: `diguy_module_net_deinitialize()`

Note: the diguyNetInterface class acts a wrapper around either a DIS or a HLA (HLA1.3/HLA1516) network interface, the type of interface that is created depends on which library is linked against.

See the "DI-Guy API, VR-Link HLA Module" chapter of the DI-Guy SDK Reference Manual for more information about these functions, as well as a listing of other HLA-related functions offered by *DI-Guy*.

If *DI-Guy* fails to join the exercise, it is generally 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`.

Controlling 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 line:

`hla_force_update = 5` in `networking.cfg`

- 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.

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* will use the same entity mapping scheme as described in the DIS model mapping sections above.

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
```

DI-Guy FOM extensions for HLA

The *DI-Guy FOM extensions* define five *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.

Attribute	Type
BaseEntity.PhysicalEntity.DIGuyAction	string
BaseEntity.PhysicalEntity.DIGuyAppearance	string
BaseEntity.PhysicalEntity.DIGuyArc	string
BaseEntity.PhysicalEntity.DIGuyType	string
BaseEntity.PhysicalEntity.DIGuyParent	string

Here is the Object Model Template (**.omt**) description of the *DI-Guy FOM extension*:

```
(Class (ID xxx)
       (Name "PhysicalEntity")
       (PSCapabilities S)
       (Description "A base class of all discrete platform scenario domain
       participants.")
       (SuperClass (BaseEntity))
       ...
       (Attribute (Name "DIGuyAction")
              (DataType "string")
              (Cardinality "1")
              (Units "(none)")
              (Accuracy "N/A")
              (AccuracyCondition "N/A")
              (UpdateType Conditional)
              (TransferAccept N)
              (UpdateReflect UR)
              (Description "Current DI-Guy Action")
              (RoutingSpace "HyperSpace")
              (DeliveryCategory "best_effort")
              (MessageOrdering "receive")
       )
       (Attribute (Name "DIGuyAppearance")
              (DataType "string")
              (Cardinality "1")
              (Units "(none)")
              (Accuracy "N/A")
              (AccuracyCondition "N/A")
              (UpdateType Conditional)
              (TransferAccept N)
              (UpdateReflect UR)
              (Description "Current DI-Guy Character Appearance")
              (RoutingSpace "HyperSpace")
              (DeliveryCategory "best_effort")
```

```
            (MessageOrdering "receive")
        )

        (Attribute (Name "DIGuyArc")
            (DataType "string")
            (Cardinality "1")
            (Units "(none)")
            (Accuracy "N/A")
            (AccuracyCondition "N/A")
            (UpdateType Conditional)
            (TransferAccept N)
            (UpdateReflect UR)
            (Description "DI-Guy Motion Arc")
            (RoutingSpace "HyperSpace")
            (DeliveryCategory "best_effort")
            (MessageOrdering "receive")
        )
        (Attribute (Name "DIGuyType")
            (DataType "string")
            (Cardinality "1")
            (Units "(none)")
            (Accuracy "N/A")
            (AccuracyCondition "N/A")
            (UpdateType Conditional)
            (TransferAccept N)
            (UpdateReflect UR)
            (Description "DI-Guy Character Type")
            (RoutingSpace "HyperSpace")
            (DeliveryCategory "best_effort")
            (MessageOrdering "receive")
        )
        (Attribute (Name "DIGuyParent")
            (DataType "EntityIdentifierStruct")
            (Cardinality "0-1")
            (Accuracy "perfect")
            (AccuracyCondition "always")
            (UpdateType Conditional)
            (TransferAccept N)
            (UpdateReflect UR)
            (Description "An entity in or on which a DI-Guy entity is
            riding.")
            (RoutingSpace "HyperSpace")
            (DeliveryCategory "best_effort")
            (MessageOrdering "receive")
        )
}
```

Troubleshooting your HLA application

- **If you are using HLA**:
  - 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 HLA (or DIS under vpNet)**, 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
MAK_VRLDIR=C:/MAK/vrlink4.0.3
MAKLMGRD_LICENSE_FILE=@accordion
```

FIRE and DETONATE interactions in DI-Guy

NOTE: In this section we use the term *interactions*, from HLA parlance, to refer to DIS fire PDUs, DIS detonation PDUs, HLA fire interactions, and HLA detonation interactions.

DI-Guy characters interact with fire and detonation PDUs when interoperating in DIS/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.

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

DI-Guy Scenario performs all 4 steps above out-of-the-box.

DI-Guy for Vega Prime performs steps 1 and 3. Users must implement steps 2 and 4.

DI-Guy Scenario issues DETONATION interactions as follows:

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

- *DI-Guy Scenario* displays muzzle flash and will issue a FIRE interaction and a DETONATION interaction (*DI-Guy* Scenario currently assumes that bullets travel instantaneously and along a straight line) 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 will be issued. If history is not being saved, during the 2-N loops DI-Guy Scenario will continue to issue FIRE and DETONATION interactions.
- *DI-Guy for Vega Prime* displays muzzle flash and will issue 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 will be issued. If history is not being saved, characters do not move and no FIRE or DETONATION interactions are issued.

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

Network configuration parameters for DI-Guy Scenario

Here is a summary of the network configuration parameters in:
`$DIGUY/custom/config/diguy/scenario/network.cfg`

These parameters can be tuned to balance performance and fidelity. Be careful to save a copy of network.cfg if you modify it so that you can return to default settings if you encounter a problem.

Name and default value	Description
`ip_address = 239.255.0.1`	Address to which outgoing DIS traffic will be 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 "bothering" every machine on the network.
`Port = 3000`	Network port used for DIS networking.
`exercise = 1`	DIS exercise number.
`site = 0`	DIS site number.
`host = 0`	DIS host number.
`velocity_smoothing_period = 0`	Time duration (in seconds) for character current velocity to match dead-reckoned velocity.
`position_smoothing_period = 0.1`	Time duration (in seconds) for character current position to match dead or live-reckoned position.
`translation_threshold = 0.05`	Permitted translation error (in meters) before DIS update is sent.
`rotation_threshold = 0.2`	Permitted rotation error (in radians) before DIS update is sent.
`timeout_interval = 12`	How long (in seconds) with out packet traffic before an entity is assumed to be no longer on the network and is removed.
`iguy_velocity_smoothing_period = 0.3`	For characters not on a path, this time constant is used for smoothing their velocity before transmitting it.
`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

`hla_force_update = 5`	**HLA only**. 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.
`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_entity_custom_pdus = 1`	**DIS only**. Indicates whether *DI-Guy* custom PDUs are to be transmitted.
`hla_diguy_fom = 1`	**HLA only**. Indicates whether *DI-Guy* custom entity attributes are to be transmitted and received.
`hla_advisories = 0`	**HLA only**. Indicate whether to activate advisories. Note this value must agree with your RTI config file.
`sync_mode_enabled = 0`	**DIS only**. 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.
`sync_master = 0`	**DIS only**. **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`	**DIS only**. **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`	**DIS only**. **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 doesn't 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`	**DIS only**. **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. E.g. 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 doesn't 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.
`pitch_for_sync_mode = 0`	**DIS only**. **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. E.g. 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 doesn't 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`	**DIS only**. A value from 220 to 255, used as the PDU kind ID for *DI-Guy* custom PDUs.
`federation_name = DI-Guy`	**HLA only**. Name of the federation execution, as well as the base name of the .FED file.
`detonation_range = 5`	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.

Database coordinates and DI-Guy networking

Positioning the terrain database on Earth's surface

DI-Guy supports geocentric, UTM, and flat earth coordinate conversions. *DI-Guy* works in database coordinates. When *DI-Guy* communicates with a simulation via 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 WGS1984 ellipsoid representation of the earth.

Setting 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()
```

See net_send/net_recv programming examples for more information.

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

Controlling DI-Guy Scenario Remotely using Custom PDUs

Custom PDUs have been developed to allow the remote control of *DI-Guy Scenario*. Once *DI-Guy Scenario* has joined the network, it will react to custom PDUs to load scenarios, merge other scenarios (vignettes) into the current scenario, set time-of-day information, perform playback control functions (start, stop, reset, etc), trigger scripts in the scenario, or even receive scripts composed by the sender. There are

also some PDUs then sent by *DI-Guy Scenario* that allow the sender to know when activities were completed.

It should be noted that the network master could also be another *DI-Guy Scenario* application if desired.

Control PDU enumeration

Please see %(DIGUY)\include\diguy_dis_constants.h for the up to date version of this data. This chart is from DI-Guy 12.5.0.

```
enum diguyCustomPDUType
{
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,
DIGUY_CUSTOM_PDU_ID_SET_CHAR_PAUSED, // (no longer used)
DIGUY_CUSTOM_PDU_ID_SET_SCENARIO_FRAME = 6,
DIGUY_CUSTOM_PDU_ID_SET_SCENARIO_PLAYBACK_MODE,
DIGUY_CUSTOM_PDU_ID_LOAD_SCENARIO_CAMERA,
DIGUY_CUSTOM_PDU_ID_RESET,
DIGUY_CUSTOM_PDU_ID_HAIL, // 10
DIGUY_CUSTOM_PDU_ID_LOAD_SCENARIO_FILE,
DIGUY_CUSTOM_PDU_ID_SET_CAMERA_DETAILS,
DIGUY_CUSTOM_PDU_ID_XC_LOAD_SCENARIO = 13, // 13
DIGUY_CUSTOM_PDU_ID_XC_LOAD_SCENARIO_COMPLETE,
DIGUY_CUSTOM_PDU_ID_XC_MERGE_SCENARIO,
DIGUY_CUSTOM_PDU_ID_XC_MERGE_SCENARIO_COMPLETE,
DIGUY_CUSTOM_PDU_ID_XC_SET_TIN_TOD,
DIGUY_CUSTOM_PDU_ID_XC_TIME_NOW,
DIGUY_CUSTOM_PDU_ID_XC_CONTROL,
DIGUY_CUSTOM_PDU_ID_XC_TRIGGER_SCRIPT, // 20
DIGUY_CUSTOM_PDU_ID_XC_EVAL_SCRIPT,
DIGUY_CUSTOM_PDU_ID_XC_SET_FF_SPEED,
DIGUY_CUSTOM_PDU_ID_XC_TRIGGER_SIGNAL,
DIGUY_CUSTOM_PDU_ID_XC_RESET_SCENARIO,
DIGUY_CUSTOM_PDU_ID_XC_ENTER_SLAVE_MODE,
DIGUY_CUSTOM_PDU_ID_XC_EXIT_SLAVE_MODE,
DIGUY_CUSTOM_PDU_ID_VARIABLE_STATE,
DIGUY_CUSTOM_PDU_ID_SAVE_SCENARIO,
DIGUY_CUSTOM_PDU_ID_SAVE_SCENARIO_COMPLETE,
DIGUY_CUSTOM_PDU_ID_AI_SEND_MESSAGE_TO_CHARACTER, // 30
DIGUY_CUSTOM_PDU_ID_AI_SEND_MESSAGE_TO_GROUP,
DIGUY_CUSTOM_PDU_ID_AI_BROADCAST_MESSAGE,
DIGUY_CUSTOM_PDU_ID_AI_BROADCAST_MESSAGE_TO_GROUP,
DIGUY_CUSTOM_PDU_ID_BEGIN_APPEARANCE_EFFECT,
DIGUY_CUSTOM_PDU_ID_END_APPEARANCE_EFFECT,
DIGUY_CUSTOM_PDU_ID_STOP_APPEARANCE_EFFECT,
```

```
DIGUY_CUSTOM_PDU_ID_SET_SCENE_OBJECT_BIT,
DIGUY_CUSTOM_PDU_ID_BROADCAST_EXERCISE_DATETIME,
DIGUY_CUSTOM_PDU_ID_REQUEST_EXERCISE_DATETIME_BROADCAST, // unused
DIGUY_CUSTOM_PDU_ID_SET_SHAPE_VARIATION, // 40
DIGUY_CUSTOM_PDU_ID_EXECUTE_GESTURE,
DIGUY_CUSTOM_PDU_ID_CHARACTER_SCALE,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED0,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED1,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED2,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED3,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED4,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED5,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED6,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED7,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED8,
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED9,
DIGUY_CUSTOM_PDU_ID_COUNT // 51
} ;
```

Description:

This enumeration lists the message IDs for DI−Guy's custom pdus. Note that most custom PDUs are useful only when DI-Guy Scenario is acting as a SAF or Lifeform Server in a distributed exercise.

DIGUY_CUSTOM_PDU_ID_SET_CHAR_TYPE
> Contents: String
> Meaning: call diguyCharacter::set_character_type()

DIGUY_CUSTOM_PDU_ID_SET_CHAR_APPEARANCE
> Contents: String A, String B
> Meaning: call diguyCharacter::set_appearance()
optionally calls
diguyCharacter::set_current_head_appearance()

DIGUY_CUSTOM_PDU_ID_SET_CHAR_ACTION
> Contents: String A, String B
Meaning: Set the character action to A, the second string is either "p" or "" (the empty string). A value of "p" for the second string indicates that the action is paused. Characters are typically paused if the sending application such as DI−Guy Scenario has been playing and the user hits "pause" on the VCR control.

DIGUY_CUSTOM_PDU_ID_SET_CHAR_VISIBLE
> Contents: Long Boolean

DIGUY_CUSTOM_PDU_ID_SET_SCENARIO_FRAME
> Contents: Long val

> Meaning: call diguyScenario::set_frame_desired(val)

DIGUY_CUSTOM_PDU_ID_SET_SCENARIO_PLAYBACK_MODE
> Contents: Long val
> Meaning: call diguyScenario::set_playback_mode(val)

DIGUY_CUSTOM_PDU_ID_LOAD_SCENARIO_CAMERA
> Contents: string
Meaning: call
diguyScenario::get_scenario_camera()−>load_settings(string)

DIGUY_CUSTOM_PDU_ID_RESET
> Meaning: call diguyScenario::reset()

DIGUY_CUSTOM_PDU_ID_HAIL

DIGUY_CUSTOM_PDU_ID_LOAD_SCENARIO_FILE
> Contents: string
> Meaning: call diguyScenario::load(string)

DIGUY_CUSTOM_PDU_ID_SET_CAMERA_DETAILS
DIGUY_CUSTOM_PDU_ID_XC_LOAD_SCENARIO
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String => Filename −
String => Reference name

DIGUY_CUSTOM_PDU_ID_XC_LOAD_SCENARIO_COMPLETE
> Contents: string
> Meaning: Indicates scenario has finished loading
Contents:
− NetU16 => Target_site − NetU16 => Target_host − NetFlt32 => Time of
Day − NetFlt32[3] => Refx, Refy, Refz − NetFlt32[3] => x, y, z −
NetFlt32[3] => w,r,p − String => Filename − String => Reference name

DIGUY_CUSTOM_PDU_ID_XC_MERGE_SCENARIO_COMPLETE
> Contents: string
> Meaning: Indicates scenario has finished loading

DIGUY_CUSTOM_PDU_ID_XC_SET_TIN_TOD
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − NetFlt32 => Time of Day
> Meaning: call diguyScenario::set_tin_time_of_day()

DIGUY_CUSTOM_PDU_ID_XC_TIME_NOW
> Contents:

− NetFlt32 => seconds since midnight − NetFlt32 => seconds to smooth
> Meaning: call diguyScenario::set_sync_info()

DIGUY_CUSTOM_PDU_ID_XC_CONTROL
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String => Command
("RUN", "STOP", "RESET", "FF", "REWIND")

DIGUY_CUSTOM_PDU_ID_XC_TRIGGER_SCRIPT
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String Script name

DIGUY_CUSTOM_PDU_ID_XC_EVAL_SCRIPT
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String => Script text

DIGUY_CUSTOM_PDU_ID_XC_SET_FF_SPEED
Contents:
− NetU16 => Target_site − NetU16 => Target_host − NetFlt32 => Speed
multiple

DIGUY_CUSTOM_PDU_ID_XC_TRIGGER_SIGNAL
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String => Signal to
trigger

DIGUY_CUSTOM_PDU_ID_XC_RESET_SCENARIO
> Contents:
− NetU16 => Target_site − NetU16 => Target_host

DIGUY_CUSTOM_PDU_ID_XC_ENTER_SLAVE_MODE
> Contents:
− NetU16 => Target_site − NetU16 => Target_host

DIGUY_CUSTOM_PDU_ID_XC_EXIT_SLAVE_MODE
> Contents:
− NetU16 => Target_site − NetU16 => Target_host

DIGUY_CUSTOM_PDU_ID_VARIABLE_STATE
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − String => variable −
String => value

DIGUY_CUSTOM_PDU_ID_SAVE_SCENARIO
> Contents:

− NetU16 => Target_site − NetU16 => Target_host − NetU16 =>
use_user_name − String => file name

DIGUY_CUSTOM_PDU_ID_SAVE_SCENARIO_COMPLETE
> Contents:
− String => file name

DIGUY_CUSTOM_PDU_ID_AI_SEND_MESSAGE_TO_CHARACTER
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − NetU16 => target AI
network number − String => sender − String => message_type − String =>
message − String => message_params
> Meaning: call
diguyCharacter::agent_accept_message(sender,
message_type,
message,
message_params);

DIGUY_CUSTOM_PDU_ID_AI_SEND_MESSAGE_TO_GROUP
> Contents:
− String => group_name − String => sender − String => message_type −
String => message − String => message_params
> Meaning: call
diguyCharacterGroup::send_message_to_all_members(sender,
message_type,
message,
message_params);

DIGUY_CUSTOM_PDU_ID_AI_BROADCAST_MESSAGE
> Contents:
− String => sender − NetFlt32 => radius − String => message_type − String
=> message − String => message_params
> Meaning: call
diguyCharacter::agent_broadcast_message(sender,
radius,
message_type,
message,
message_params);
Where character is the sender of message.

DIGUY_CUSTOM_PDU_ID_AI_BROADCAST_MESSAGE_TO_GROUP
> Contents:
− String => sender − String => group_name − NetFlt32 => radius − String
=> message_type − String => message − String => message_params
Meaning: call

diguyCharacter::agent_broadcast_message_to_group(group_name,
radius,
message_type,
message,
message_params);
Where character is the sender of message.

DIGUY_CUSTOM_PDU_ID_BEGIN_APPEARANCE_EFFECT
> Contents:
− String => appearance effect − bool => flag if theres extended data
> Optional extended data:
− NetFlt32 => override scale − NetFlt32[3] => override_offset x, y, z −
NetFlt32 => override duration
> Meaning: call
diguyCharacter::begin_appearance_effect(app_name, link_name, override_scale,
override_offset_x, override_offset_y, override_offset_z,
override_duration,
or
diguyCharacter::begin_appearance_effect(app_name);

DIGUY_CUSTOM_PDU_ID_END_APPEARANCE_EFFECT (Gradual Stop)
> Contents:
> String => appearance_effect_name
− String => link_name
> Meaning: call
diguyCharacter::end_appearance_effect(appearance_effect_name,link_name)

DIGUY_CUSTOM_PDU_ID_STOP_APPEARANCE_EFFECT (Instant Stop)
> Contents:
> String => appearance_effect_name
− String => link_name
> Meaning: call diguyCharacter::stop_appearance_effect(appearance_effect_name, link_name)

DIGUY_CUSTOM_PDU_ID_SET_SCENE_OBJECT_BIT
> Contents:
> bool => true false
> Meaning: call
diguyCharacter::set_is_scene_object(value)

DIGUY_CUSTOM_PDU_ID_BROADCAST_EXERCISE_DATETIME
internal use only
> Contents:
− NetU16 => Target_site − NetU16 => Target_host − NetU16[7] => year,
month, day, hour, minute, second, msec
> Meaning: call

bdiScenario::set_network_datetime(network_datetime);

DIGUY_CUSTOM_PDU_ID_REQUEST_EXERCISE_DATETIME_BROADCAST
Unused

DIGUY_CUSTOM_PDU_ID_SET_SHAPE_VARIATION
> Contents:
complicated: a series of NetU16 (link index), NetU16 (shape index) followed by a slightly compressed chunk of pixel data. Should probably only happen once per character.

```
int offset = 0;
short link_index = 0;
while(pdu−>pullShort(link_index, offset) == 0)
{
    short shape_index = 0;
    pdu−>pullShort(shape_index, offset);
    bdiLink * link = character−>get_link_at_index(link_index);
    if(link)
    {
        bdiShape * shape = link−>get_shape(shape_index);
        if(!shape−>get_unique_ramp_data())
        {
            const unsigned char * data_blob = NULL;
            int bytes_read;
            if(pdu−>pullUnsignedData(data_blob, offset, bytes_read) == 0)
            {
                unsigned char * data_blob_copy = new unsigned char [shape−>get_unique_int q;
                for(q = 0; q < bytes_read/3; q++)
                {
                    data_blob_copy[q*4] = data_blob[q*3];
                    data_blob_copy[q*4+1] = data_blob[q*3+1];
                    data_blob_copy[q*4+2] = data_blob[q*3+2];
                    data_blob_copy[q*4+3] = 255;
                }
                // fill the rest of the buffer
                for( ; q < shape−>get_unique_ramp_data_size(); q++)
                {
                    data_blob_copy[q*4] = 128;
                    data_blob_copy[q*4+1] = 128;
                    data_blob_copy[q*4+2] = 128;
                    data_blob_copy[q*4+3] = 255;
                }
                shape−>set_unique_ramp_data(data_blob_copy);
            }
        }
    }
}
```

DIGUY_CUSTOM_PDU_ID_EXECUTE_GESTURE
> Contents:
− String => gesture_name − Long => reps − NetFlt32 => overall_duration −
NetFlt32 => channel_A_weight
> Meaning: call
diguyCharacter::execute_gesture(gesture_name,reps,
overall_duration, channel_A_weight);

DIGUY_CUSTOM_PDU_ID_CHARACTER_SCALE
> Contents:
− NetFlt32[3] => scale x,y,z
> Meaning: call diguyCharacter::set_scale(x,y,z);

DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED0
.
.
.
DIGUY_CUSTOM_PDU_ID_PLUGIN_RESERVED9

7. 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, etc.

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
- facilitate external geometry file loaders
- etc.

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 (e.g. – Open Scene Graph, Vega Prime) and immediate mode renderers (e.g. - 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 Open Scene Graph, OpenGL, and DirectX9 delivered with DI-Guy, to guide their development of a DI-Guy Graphics API solution tailored to their specific rendering situation.

8. Miscellaneous

Complete Vehicles

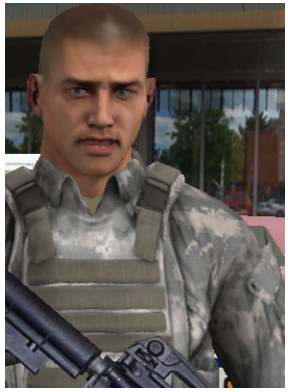
Complete vehicle character types have the following capabilities:

- Rolling and turning wheels.
- Smooth acceleration and deceleration.
- Doors that open and close.
- Body that can rock.
- Aim-able turret.

Use character type “vehicle_09” to access complete vehicles.

See the DI-Guy Scenario user manual for more information about Complete Vehicles

Expressive Faces



Expressive Faces are an optional module available for DI-Guy. Expressive Faces are animated head graphics that can portray different emotional states, blink, and do lip-sync to speech. To activate the expressive faces module, you need to to initialize and deinitialize the expressive face module and have a valid expressive faces license.

DI-Guy Expresive Faces has recently been upgraded to FaceFX technology, see the Expressive Faces User Guide for more information.

DI-Guy Version History for Windows/Linux

DI-Guy has extensive cross platform support, with versions available for Windows and Linux. The following contains specific version information for DI-Guy versions 7 through 11.

DI-Guy 12 for Windows 32/64 and Linux 64 was released in the June 2012. It compiles with MSVC 9 & 10 and gcc 4.1.2 and supports Vega Prime 5.0 in August 2012.

DI-Guy 11 for Windows 32/64 and Linux 32/64 was released in the spring of 2011. It compiles with MSVC 8 & 9 and gcc 4.1.2 and supports Vega Prime 4.0.

DI-Guy 10 for Windows was released in March 2009. It supports OpenGL, DirectX9, and OSG. Compiles with MSVC 7 & 8. Supports Vega Prime 3.0. A Windows 64 version is also available.

DI-Guy 10 for Linux was released in May 2009. It supports OpenGL, VegaNT/Performer, OSG. First 64 bit solution in addition to standard 32-bit. Compiles with gcc 4.1.2. Supports Vega Prime 3.0.

DI-Guy 9.0.3 for Windows, the last DI-Guy 9 release, came out in July 2008. It supports OpenGL, DirectX9, OSG. First DI-Guy AI release. Compiles with MSVC 7 & 8. Supports Vega Prime 2.2 with vpNET 2.2.

DI-Guy 9.0.1 for Linux, the last DI-Guy 9 release that compiles for gcc 3.4, came out in March 2008. It supports OpenGL, VegaNT/Performer, and OSG. Supports Vega Prime 2.2

DI-Guy 9.0.5 for Linux, the only DI-Guy 9 release that compiles for gcc 4.1.2, was released in February 2008. It supports OpenGL, VegaNT/Performer, and OSG. No Vega Prime support.

DI-Guy 8.0.6 for Windows, the last DI-Guy 8 release, came out in June 2007. It supports OpenGL, DirectX9, OSG. Compiles with MSVC 6 & 7. Supports Vega Prime 2.1 with vpNET 2.1.1.

DI-Guy 8.0.5 for Linux, the last DI-Guy 8 linux release, came out in May 2007. It supports OpenGL, VegaNT/Performer, and OSG. Compiles with gcc3.4. Supports Vega Prime 2.1.

DI-Guy 7.0.9 for Windows, the near-last version of DI-Guy 7. It supports OpenGL, DirectX8, DirectX9, VegaNT/Performer 3.7.1. Compiles with MSVC 6 & 7. Supports Vega Prime 2.0 with vpNET 2.0. **DI-Guy 7.0.11 for Windows** was a special release that supports Vega Prime 2.0 with vpNET 2.1.

DI-Guy 7.0.5 for Linux, the last DI-Guy 7 release. It supports OpenGL, Performer 3.1, and OSG. Compiles with gcc 3.2. Supports Vega Prime 2.0.

NOTE: *Special Effects* are only supported directly in DI-Guy Scenario and are mapped to Vega Prime effects on Windows.

NOTE: *OpenAL sound* is only formally supported in DI-Guy Scenario and DI-Guy for Windows.

NOTE: *Performer* no longer supported as of version 8.

NOTE: *Vega "Classic"* no longer supported as of version 8. Users encouraged to use DI-Guy Graphics API.

NOTE: Each version of DI-Guy only supports one version of *Vega Prime*. Check details above.

Appendix A: Installing DI-Guy on Windows

Installing DI-Guy

To install *DI-Guy* :

1. Install the ***DI-Guy Applications*** CD.

2. Follow the instructions that appear on your screen, acknowledging any messages that appear during the process.

3. Install the ***DI-Guy Data*** CDs

4. Follow the instructions that appear on your screen, acknowledging any messages that appear during the process.

5. Run the DI-Guy License Tool. This is available from the Start Menu. Instructions are included in the license tool for how to obtain a license from Boston Dynamics.

Advanced Topic: Managing multiple versions of DI-Guy on Windows

If for some reason you need to have multiple versions of *DI-Guy* on your machine, you can achieve your goal by careful management of your `DIGUY` environment variable and your `PATH` environment variable. It is often useful to create a script on a multiple installation system to handle clearing and setting these two critical environment variables.

To install *DI-Guy Applications* without uninstalling the existing *DI-Guy*, choose "no" when the installer says "Previous DI-Guy Applications installation detected. Uninstall?" Choose "OK" in response to "Leave current installation alone and continue? ..." Chose a folder for installation that will not overwrite an installation you wish to preserve. You will get a warning that "There is already a system environment variable named DIGUY, ..." Dismiss the warning by clicking "OK".

To install *DI-Guy Data* without uninstalling the existing *DI-Guy*, choose "no" when the installer says "Previous DI-Guy Data installation detected. Uninstall?" Choose "OK" in response to "Leave current installation alone and continue" query. Make sure to install the Data in the same folder with the associated DI-*Guy Applications*.

DI-Guy environment variable

DI-Guy applications find data and other files by looking up the environment variable `DIGUY`. If *DI-Guy* is installed in "C:\DI-Guy-12.5.0", for example, the DIGUY environment variable will be C:\DI-Guy-12.5.0.

Default setting and location

The *DI-Guy SDK* Windows installer automatically sets the DIGUY variable to the correct value and adds a *DI-Guy* folder to the path.

In the standard Windows command processor, `%DIGUY%` refers to the value of the environment variable `DIGUY`.

Changing the default variable value

To change the value of this variable, in a command shell, enter:

```
prompt> set DIGUY=c:\Another-DI-guy
```

To change the environment for newly started command shells and for programs that run by clicking **Windows Explorer** icons, use the `Control Panel System` applet.

Appendix B: Installing DI-Guy on Linux

Linux system requirements

Different versions of *DI-Guy* are built to run with particular versions of the gcc compiler, often associated with a particular build of Red Hat Enterprise. Consult the *DI-Guy* website http://www.diguy.com if you are unsure if you have the proper version of DI-Guy that corresponds to your target gcc compiler version.

DI-Guy Linux distributions come with an associated README file. Please consult this file for more information about installing the particular version of DI-Guy for Linux.

Appendix C: DI-Guy Licensing

Instructions for how to install and test DI-Guy licenses are found in readme files included in the DI-Guy distribution. To get started, first examine the main readme file in the $DIGUY/flexlm directory.

These are three basic types of licenses supported by *DI-Guy*. Ask your DI-Guy salesperson for what type of license you have purchased or are evaluating.

- **Node-locked** license

  A node-locked license ties the license to the MAC (Media Access Control) address of a particular computer. The MAC address is either an eight or twelve hexadecimal digit number.

A typical feature line in the license file looks like this:

```
FEATURE diguy_runtime bdilm 5.000 permanent uncounted 767B3A5321BE \
VENDOR_STRING=DI-Guy HOSTID=003048828c98
```

- **Dongle** license

A dongle license ties the license to a hardware dongle called a FLEXID that goes in the USB port of a Windows computer. The USB port FLEXID hostid consists of a string "FLEXID=9" followed by eight hexadecimal digits. You must run

`%DIGUY%\3rdparty\flexlm\diguy_flexlm_10.1_win32\FLEXidInstaller.exe` to install a device driver so that *DI-Guy* can read the Hostid from the dongle. Install the flexlm device driver before inserting the dongle into the computer.

A typical line in the license file looks like this:

```
FEATURE diguy_runtime bdilm 5.000 permanent uncounted 264B47B0CCA9 \
 VENDOR_STRING=DI-Guy HOSTID=FLEXID=6-a630b4f1 \
 START=16-mar-2004
```

- **Floating** license

Instead of tying the license directly to the local hardware of a computer or dongle, a floating license points to a license server computer which grants the license. The license server computer must run license managing software and have its own license file for specifying how many licenses will be floated. Any machine on the local area network can then be granted the license.

A typical client license file looks like this:

```
SERVER Bulldog-2 0052b3aa3e3d
USE_SERVER
```

The first few lines of a typical server license file looks like this:

```
SERVER Bulldog-2 0052b3aa3e3d
VENDOR bdilm
FEATURE diguy_runtime bdilm 5.000 permanent 2 E56E937C57EB \
 VENDOR_STRING=DI-Guy DUP_GROUP=H
```

Appendix D: DI-Guy Programming Examples

DI-Guy offers extensive sample programs to help get you started. Examples are found in the $DIGUY/programming_examples directory.

The contents of the DI-Guy programming examples for version 12 Windows is as follows:

OpenGL Examples – this is the main programming example directory. It contains not only examples of working with the DI-Guy OpenGL solution, but also 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.

Other:

Diguy Networking – SDK examples of broadcasting and receiving human characters for DIS, HLA 1.3, and HLA 1516. The examples use VT-MAK's VR-Link networking solution (sold separately), but are intended to assist developers with networking DI-Guy with other 3rdparty and customer proprietary networking solutions.

Plugins – Users 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 plugin software that supplements DI-Guy Scenario.

DI-Guy Digest – The DI-Guy digest is a .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.

Vega Prime Support is also available by installing the *DI-Guy for Vega Prime* application found in the vpDiguy directory of the *DI-Guy* Applications install disc.

OpenGL serves as *DI-Guy's* "default" for programming examples. This means that the OpenGL directory contains a far more extensive set of examples than the other four directories. This doesn't mean that functionality shown in the OpenGL examples wouldn't work on the other platforms. In fact, the opposite is true.

> Don't ignore the OpenGL examples! Even if you are not using the *DI-Guy* OpenGL renderer, don't 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.

“Simple” Example – our simplest example. Instantiate a DI-Guy soldier character and command him to walk, walk in a crouched fashion, go prone, and fire his weapon.

“Simple Playback” Example – our other simplest example. – play a .dss file as output from DI-Guy Scenario.

“**Shadow Playback” Example** - DI-Guy playback with different shaders.

“Guide” Example – An excellent demonstration of various DI-Guy guide algorithms in action. Remember, a guide is useful when a character’s position is being driven intermittently from an external source.

“AI Playback” Example – Similar to simple playback but with DI-Guy AI also activated when the scenario was built in DI-Guy Scenario.

“AI Populate” Example – Use DI-Guy AI completely through the SDK.

FaceFX Example – a demonstration of DI-Guy Expressive Faces via the SDK.

“Load Manager” Example – 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.

“Culling” Example – Shows culling of characters based on camera frustum. Includes demonstrates using far positioning.

“LODs Example” – Shows DI-Guy Level-of-Detail being activated, including a demonstration using far positioning.

“Path API” Example – a demonstration of some simple path commands including setting waypoints, action beads, actions, and script beads on a path.

“Pose” Example – 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.

“Gaze” Example – Demonstration of the majority of the Gaze API for commanding characters to look at thing.

"Get Position" Example – A simple example of querying DI-Guy characters for information.

"Intersection" Example – Demonstrates vector intersection to DI-Guy characters.

"Rolling Pitching Ship" – DI-Guy characters parented to a moving ship maintaining upright Z posture.

"View Settings" Example – demonstrate various methods of setting the camera, fog, and light.

"View Far Settings" Example – 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.

The *DI-Guy Graphics API* Examples – demonstrates how to use the Graphics API so that you can render into an unsupported graphics environment. 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 Examples – 6 examples showing broadcasting and receiving from DIS, HLA1.3, and HLA1516 with VT-MAK's VR-Link software.

Plugin Examples – demonstrates how to create your plugin, most typically used as an add-on module to DI-Guy Scenario.

DI-Guy Digest – The DI-Guy digest is a .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.

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

Appendix E: Transitioning to DI-Guy 12

DI-Guy 12 includes a major upgrade to its character graphics capabilities. This upgrade has required a number of non-backwards compatible changes in the graphics API.

Shader uniform binding

In order to support skinned characters that have multiple sub-skeletons (e.g. – characters using expressive faces), skinned shader uniforms have been restructured. Instead of binding shader uniforms once per character, shader uniforms are bound once per each skinned mesh/shape. Most older character appearances will not see a difference in number of shader operation executed, as they have only one sub-skeleton. DI-Guy API functions related to shader uniform binding have been modified to support this functionality.

Shader Technique

Prior to DI-Guy 12, DI-Guy character appearances had a one-to-one correspondence with a particular shader. DI-Guy 12 instead specifies a *shader technique* per character (this enables the use of Shader LODs) and the character is drawn using one of a number of possible shaders described in the technique depending on the target quality level.

The Shader Binding and Shader Technique changes have meant the removal of much of diguyGraphicsShaderInstance functionality. The following diguyGraphicsShaderInstance virtual functions calls should be upgraded by the user to the new diguyGraphicsShaderProgram virtual functions:

Old:

```
diguyGraphicsShaderInstance::bind_link_matrices()
diguyGraphicsShaderInstance::update_link_matrices()
diguyGraphicsShaderInstance::bind_color_array()
```

New:

```
diguyGraphicsShaderProgram::bind_link_matrices();
diguyGraphicsShaderProgram::update_link_matrices();
diguyGraphicsShaderProgram::bind_color_array();
```

Old:

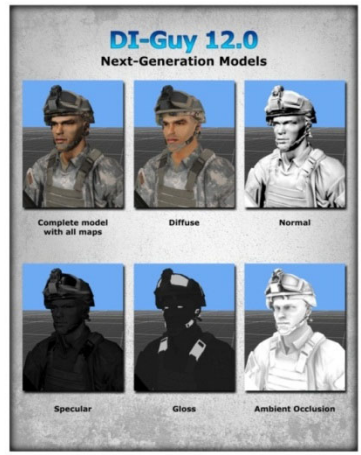
```
diguyGraphicsShaderInstance::get_num_link_matrices();
```

New:

```
diguyGraphicsShape::get_shader_matrix_data(int* num_matrixes,
 const float** matrix_data);
```

Note that diguyGraphicsShape::get_shader_matrix_data() can be called every update and sent to scene graph renderers by overriding the virtual function diguyGraphicsShape::update().

Multitexturing and Texture Binding



DI-Guy 12 characters use multiple textures for improved graphics appearance while maintaining high performance. The following function prototype has been modified to support multitexturing:

Old:

```
virtual void diguyGraphicsTexture::bind_now();
```

New:

```
virtual void diguyGraphicsTexture::bind_now(diguyGraphicsTextureMapType texture_type);
```

The `diguyGraphicsTextureMapType` is an integer that maps to the texture unit the texture should be in. DI-Guy sample shader code demonstrates how to set up this uniform value. A simple lookup table can be added to the code to map the diguyGraphicsTextureMapType index to the user's preferred texture unit.

Multitexturing and Compressed texture maps

DI-Guy 12 characters use multiple textures for improved graphics appearance; it is therefore strongly recommend that those textures are saved and loaded via DI-Guy's compressed texture file format to maintain high performance. This format takes RGB and PNG source data and compresses them to DDS textures using NVIDIA texture tools and then performs a lossless compression pass with lz4 http://code.google.com/p/lz4/ which cuts disk usage by 30%. This compression may take a few seconds but is a one-time event unless the source files are changed. The DIGuyGeometryProcessor executable can be used to force the creation of all of the DDS compressed textures at once. Once textures are placed into compressed format the source data is no longer required. **DI-Guy ships with the textures already compressed.** Normal Map compression is done into the DXT5_nm texture format which rearranges the

texture components. If compression isn't enabled normal maps **will not** work without changes to the default shaders.

Linear Lighting and Gamma Correction

DI-Guy now defaults to using sRGB texture formats (this can be changed using a flag in the graphics init structure). This allows for gamma correct lighting, which improves the appearance of characters. More can be read about linear lighting in *GPUGems3 -The Importance of Being Linear* (http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html).

If the target shader doesn't support sRGB texture formats, removal of the gamma correction factor is recommended. The #define GAMMA_FACTOR should be set programmatically as part of shader compilation. The gamma correction factor code is found at the end of most DI-Guy fragment shaders:

```
gl_FragColor.rgb = pow(color4.rgb, vec3(1.0 / GAMMA_FACTOR));
```

Alternatively users can do a gamma correction for linear shaders in engines that don't support sRGBs using texture samplers. For example:

```
gl_FragColor.rgb = pow(sampler.rgb, vec3(2.2));
```

Note that the OSG versions for which DI-Guy provides programming examples do not support sRGB textures. To avoid a performance penality in texture sampling sRGB support is disabled in those examples and the GAMMA_FACTOR has been set to 1.0.

Informing DI-Guy of shader-supported texture maps

Target shaders may not support all the various texture maps used by DI-Guy. Users can inform DI-Guy about which subset of texture maps are supported by calling:

```
diguyOglGraphicsShaderProgram::set_textures_used_bitmask()
```

See `%DIGUY%\programming_examples\diguy_graphics_api\ogl_examples\common\diguyOglGraphicsShaderProgram.cpp` for an example. This immediate mode rendering solution will not attempt to bind any textures that don't bitwise-and (&) with the current shader's bitmask. DI-Guy 12 sample rendering code has been updated to show enabling and disabling various parts of the bump mapped shader on a per mesh basis. Some DI-Guy skinned character appearances have not been upgraded with bump maps. The shader technique code will automatically shift to per pixel lighting for these non-bump mapped appearances.

Shader LODs

Shader LODs (Level of Detail) can be controlled at the per character level or via the functions:

```
diguyApp::set_max_character_shader_quality_level()
diguyApp::set_max_scene_object_shader_quality_level()
```

Linear Draw Pipeline is now the default

DI-Guy 11.0.2 introduced the concept of a *linear draw pipeline* which avoided pushing and popping the OpenGL matrix stack recursively while drawing segmented portions of the character and instead sent pre-calculated matrices to the video card.

The linear draw pipeline has many advantages:

- The matrices were already being calculated, their use is essentially "free".
- No need to traverse the character link hierarchy pushing and popping the OpenGL stack multiple times to draw a segmented geometry (e.g. – a hand) attached to a skinned body.
- The previous default solution used deprecated OpenGL functions.
- It was not possible to support Normal Map character graphics with the previous solution because the Normal Map rendering technique requires knowing where the camera and lights are in relation to the character's modelview matrix.

The new, simpler draw pipeline opens options for future draw instancing developments.

Simplified parented character drawing

DI-Guy 12 has simplified the drawing of parented characters; most of the complexity is handled by internal update code. Drawing parented characters is now identical to drawing unparented characters.

Shader gl_Position change

The new bump map shader requires that rigid portions of the character be lit in the same space as the character. DI-Guy has added a new uniform `ufrm_link_matrix` that represents the rigid object link transform relative to the character. For consistency all DI-Guy 12 reference shaders have been updated.

Old:

```
gl_Position = ftransform();
```

New:

```
gl_Position = gl_ModelViewProjectionMatrix * ufrm_link_matrix * gl_Vertex;
```

Data driving shaders via cfg files

Prior to DI-Guy 12, DI-Guy required that some render-specific data be enumerated in shader configuration files. DI-Guy 12 favors procedurally determining that data by querying the underlying OpenGL shaders. End users should not be affected since the API for accessing that data in the graphics API was fairly minimal.

How to leverage DI-Guy bump map support

Users are encouraged to examine the file `%DIGUY%\programming_examples\diguy_graphics_api\ogl_examples\common\diguyOglGraphicsLink.cpp` to learn how to leverage DI-Guy bump maps. In the begin_character_draw() function, the main scene light and camera needs to be placed into the local space of the character. Users should query the target scene graph to get that information. DI-Guy 12 provides an inverse rotation matrix accessor to facilitate this. DI-Guy 12 programming examples set reasonable default values for the diguyScenario camera and primary light so that queries will return useful information.

Rigid and Skinned Models supported with Collada - New vertex formats

DI-Guy 12 supports both rigid and skinned Autodesk 3ds Max meshes exported as a Collada file. These two new formats are:

DIGUY_GRAPHICS_VERTEX_FORMAT_V3_N3_T2_T4_I4F_W4 (skinned)
DIGUY_GRAPHICS_VERTEX_FORMAT_V3_N3_T2_T4 (rigid)

The new skinned format replaces the older skinned vertex format. Users are encouraged to support both vertex formats as well as the DIGUY_GRAPHICS_VERTEX_FORMAT_V3_N3_T2 suitable for OpenFlight files. Note that the new geometry meshes include an additional 4f vector representing tangent data (3f) and UV mirroring (1f). These values are generated when a dae file is loaded and saved as part of the DI-Guy binary geometry (.bdg) format. The new formats result in a small increase in the memory footprint; however the number of unique meshes isn't large.

Appendix F: Transitioning to DI-Guy 12.5

C++ callback:

float AltitudeFunction(diguyScenario* scenario, float x, float y, float old_z);
has been replaced with
float AltitudeFunction(diguyScenario* scenario, float x, float y, float old_z, int * valid);
code will need to be updated before recompiling.

Graphics API

- bdiGraphicsInitGraphicsAPI.use_4x3_shader_matrices must be set to 0 if you aren't planning on upgrading shaders to use 4x3 matrices.
- As a performance optimization we are no longer transposing our internal matrix representation in C++ before sending to OpenGL. 4x4 matrix bind should get updated to

```
glUniformMatrix4fv(m_bind_location_ufrm_link_matrices,
    num_mats,
    !get_transposed_matrix_data(), // whether to transpose matrix
    mat_data); // pointer to matrix data
```

where bdiGraphicsInitGraphicsAPI.transpose_shader_matrices has been properly set see diguyOglGraphicsShaderProgram.cpp for example.