



DI-Guy AI 12.5
USER GUIDE

DI-Guy AI 12.5 User Guide

© 2013 Boston Dynamics
All rights reserved

Boston Dynamics
78 Fourth Avenue
Waltham, MA 02451 USA

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

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software in any fashion or on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this document may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without prior written consent of Boston Dynamics. Information in this document is subject to change without notice and does not represent a commitment on the part of Boston Dynamics.

OpenGL is a registered trademarks of Silicon Graphics.

OpenFlight is a registered trademark of Presagis.

Visual C++, *Windows2000/XP*, and *Direct3D* are registered trademarks of Microsoft Corp.

FLEXlm is the registered trademark of GLOBEtrout Inc and/or Macrovision.

Certain models were provided by CG², Presagis, Antycip, Engineering and

Computer Simulations, Inc., and Viewpoint DataLabs International.

VR-Link is the registered trademark of MÄK Technologies © 1997-2013 www.mak.com

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

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

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

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

OpenAL is a trademark of Creative Labs, Inc.

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

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

Lua © 1994-2013 Lua.org, PUC-Rio www.lua.org

pthread sourceware.org/pthreads-win32

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

Part No. DI-Guy AI 12.5.0-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 AI models the morning commute at Penn Station, NYC.

Table of Contents

1. WELCOME TO DI-GUY AI.....	8
2. DI-GUY AI TUTORIALS	15
2.1 TUTORIAL 1 – QUICKSTART WITH DI-GUY AI MINDS – AI WORLD CREATION.....	15
2.1.1 Create a civilian population with a simple pattern of life	16
2.1.2 Create soldiers that use formations.....	18
2.1.3 Create an enemy squad and initiate combat.....	20
2.1.4 Try loading your terrain into AI World Creation	20
2.2 TUTORIAL 2: QUICKSTART WITH BASE BEHAVIORS	21
2.2.1 Create a Crowd using Base Behavior Travel.....	22
2.2.2 Add a Wander Crowd	24
2.2.3 Trigger a flee base behavior.....	25
2.3 TUTORIAL 3 – EXPLORE OTHER SAMPLE AI SCENARIOS INCLUDED WITH DI-GUY	27
2.3.1 AI Crosswalks.....	27
2.3.2 AI Schedule.....	28
2.3.3 AI Train Station Alarm	29
2.3.4 AI Multi-Floor Path Planning	29
2.3.5 AI Middle East Town	32
2.4 TUTORIAL 4: CREATING A AI SUBCLASS (PACKAGE) FROM SCRATCH.....	33
2.5 TUTORIAL 5: FROM CHARACTER TO AGENT, STEP BY STEP	36
2.6 TUTORIAL 6: CREATE A NAVMESH	41
3. DI-GUY AI CONCEPTS	42
3.1 AGENTS AND CROWDS	42
3.2 CROWD PROFILES.....	43
3.3 AGENTS AND BEHAVIORS.....	44
3.3.1 Travel Behavior.....	45
3.3.2 Path Follow Behavior.....	46
3.3.3 Wander Behavior.....	46
3.3.4 Mingle Behavior	47
3.3.5 Flee Behavior	47
3.3.6 Pursue Behavior	48
3.3.7 Attack Behavior.....	48
3.3.8 Idle Behavior	49
3.3.9 None Behavior.....	49
3.4 REGIONS AND SUB-REGIONS	50
3.5 ACTION SELECTION.....	52
3.5.1 Speed Types	52
3.5.2 Speed Zones.....	52
3.5.3 Desired Posture and Desired Variant	57
3.6 OBSTACLES OVERVIEW.....	59
3.6.1 Static Object Sensing and Avoidance	59
3.6.2 Feeler Static Avoidance Method.....	60
3.6.3 Disabled Static Avoidance Method.....	61

3.6.4	<i>Dynamic Object Sensing and Avoidance</i>	61
3.6.5	<i>Repulsion Zones Dynamic Avoidance Method</i>	62
3.6.6	<i>Repulsion Zones and Vehicles</i>	63
3.6.6.1	<i>Disabled Dynamic Avoidance Method</i>	64
3.7	DETECTING AND GETTING OUT OF TRAPS	64
3.7.1	<i>Trap Detection</i>	64
3.7.2	<i>Untrap Methods</i>	65
3.8	INPUT MODE PALETTE	66
3.8.1	<i>Crowd Blitzer</i>	67
3.8.2	<i>Crowd Editor</i>	69
	Crowd Editor Painting Parameters	70
	Crowd Editor Crowd Population Parameters	71
	Crowd Editor Behavior Paths	71
	Painting Hot Keys	72
3.8.3	<i>Region Painter</i>	73
	Region Painter Painting Parameters	73
	Painting Hot Keys	74
3.9	CROWD PROFILE	75
3.9.1	<i>Populate Settings Tab</i>	76
	The Populate Entries Table	76
	Entry Selection Method.....	77
	Default Crowd Size	77
3.9.2	<i>Default Agent Parameters</i>	77
3.10	CROWD	77
3.10.1	<i>Crowd Members Tab</i>	78
	The Crowd Member Select Box	79
3.10.2	<i>Companion Crowds Tab</i>	80
3.10.3	<i>Avoidance Methods Tab</i>	81
3.10.4	<i>Events Tab</i>	82
3.10.5	<i>The Misc Tab</i>	83
3.11	CROWD MEMBER AGENT PARAMETERS (CROWD AND CROWD PROFILE)	83
3.11.1	<i>Behavior Settings Tab</i>	83
3.11.2	<i>Behavior Settings Descriptions</i>	85
3.11.3	<i>Prop and Scene Object Avoidance</i>	87
3.11.4	<i>Character (Dynamic Object) Avoidance</i>	89
3.11.5	<i>Trap Settings and Path Planning</i>	90
3.12	CROWD LOAD MANAGER	ERROR! BOOKMARK NOT DEFINED.
3.13	THE REGION PAGE	91
3.14	SCHEDULE EDITOR	92
4.	VISIBILITY – CROWD AND AGENT-RELATED SETTINGS	94
4.1	VIEW SETTINGS – CAMERA TAB – CROWD-RELATED SETTINGS.....	97
4.2	AUTOMATIC PATH PLANNING	98
4.3	PATH PLANNING VIA THE API	99
4.4	BACKGROUND PATH PLANNING	101
5.	DI-GUY AI MINDS	104

5.1	THE DI-GUY LUA DOCUMENTATION FILES.....	104
5.1.1	DI-Guy Lua Modules.....	105
5.1.2	DI-Guy Lua Classes	106
5.1.3	DI-Guy Lua Files.....	107
5.2	UNDERSTANDING THE BASIC ARCHITECTURE OF A DI-GUY AI MIND	108
5.3	AI MIND EXAMPLE – THE SOLDIER_TRAVEL_LUA_MIND	109
5.4	THE DI-GUY LUA CODE EDITOR.....	111
5.5	LUA PACKAGES	115
5.6	VISUALIZING THE STATE MACHINE.....	119
5.7	LUA SIMPLE SOLDIER PACKAGE: A SECOND LOOK.....	120
5.8	UNDERSTANDING HOW DI-GUY AI MINDS SLEEP.....	127
5.9	DEBUGGING MINDS.....	129
5.10	MIND REFERENCES	131
6.	DI-GUY AI FAQ	132

1. Welcome to DI-Guy AI

Intelligent Simulations Need Intelligent People

Real-time simulation of virtual worlds with life-like humans is an invaluable tool for training, visualization, and planning. Modeling the behavior of hundreds or thousands of humans in a battlefield, village, marketplace, or evacuation setting can be a daunting task. Life-like simulations require diverse sets of virtual people of different sizes, shapes, and desires. Not only must they move realistically, performing tasks central to the simulation, they also need to have the critical intelligence to react realistically to ongoing changes in the world.

Example Applications requiring Intelligent Human Agents

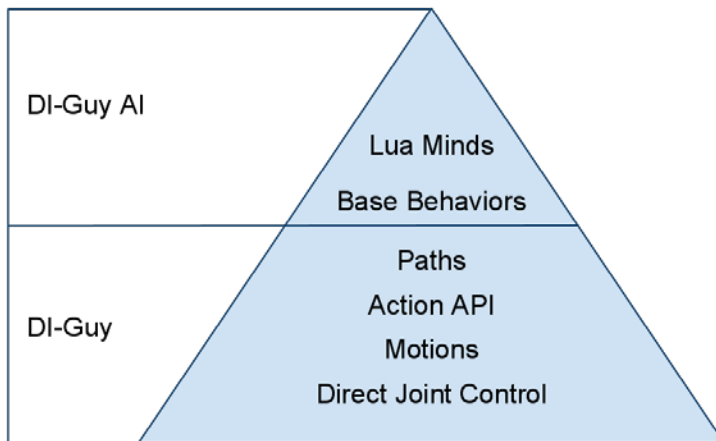
- urban combat training
- mission planning
- law enforcement training
- first responder training
- site security planning
- architectural visualization
- driving simulators
- marketing

DI-Guy AI - The solution for intelligent human simulation

DI-Guy AI makes it possible to quickly populate real-time simulations with hundreds of diverse, intelligent characters with as little as a single mouse stroke. Each character knows how to move about the space, avoid collisions, and go about their business automatically. The character's behavior is based on intelligence, beliefs, and motivations that are commanded through an easy to use graphical interface and lua scripts. DI-Guy AI provides the intelligent characters needed for modern simulations – and the architecture and infrastructure to extend that intelligence to fit exact program requirements.



A crowded marketplace is modeled with DI-Guy AI. Soldiers, insurgents, vehicles, and terrains are all standard elements *that come with DI-Guy*.



DI-Guy AI: the Top of the DI-Guy Behavior Pyramid

DI-Guy AI builds on the classic DI-Guy methods for character control such as paths and actions. DI-Guy AI starts with Base Behaviors. DI-Guy Base Behaviors are simple and understandable paradigms for specifying the behavior of a crowd, enabling easy instantiation of groups of characters that autonomously wander, travel through the terrain, flee from a person or place, follow other entities, mingle, or even attack other entities. The pinnacle of the pyramid is Lua Minds. The Minds have a hierarchical finite state machine (HFSM) architecture described by a set of open user-modifiable lua scripts.

DI-Guy AI builds on DI-Guy

DI-Guy AI is part of the DI-Guy suite of software tools for human simulation. DI-Guy AI works as a layer on top of either DI-Guy Scenario or DI-Guy SDK. This means, above all, a full cast of life-like people and a rich API to control them. DI-Guy characters look and move realistically; making natural transitions from one activity to the next while responding to high-level direction. DI-Guy avatars are completely operational and ready to go upon installation. They go where they are told to go, performing tasks and actions as directed by the author. Each character has its own skeleton, texture-mapped geometry, behavior library, and real-time motion engine.



DI-Guy AI Characters are Agents

DI-Guy AI introduces the concept of a character as an *Agent*. An agent is a DI-Guy character that uses AI to self-determine his or her motion. An agent is aware of the buildings, obstacles, and other people in its environment, so that it can make decisions about how to move through the scene in a logical and realistic manner. Agents react intelligently and automatically to potential collision events (there is a car approaching...get out of the way), asynchronous events (a shot was fired nearby...flee), and messages ("Follow me!").

Agents belong to larger aggregate entities called Crowds. Crowds allow for groups of agents to be controlled collectively for fast authoring and runtime control. An individual agent can always be controlled directly as well.

Base Behaviors

DI-Guy AI includes Base Behaviors such as *Wander*, *Mingle*, *Follow*, *Attack*, *Flee*, and *Travel*. These behaviors describe how the agents operate in their world in a clear and simple manner that is easy for the author to understand. The agents perform collision avoidance with other agents, stationary obstacles, and larger moving objects such as vehicles – and do so in a high performance manner so that thousands of agents can populate a world simulated on a single computer.

Path Planning

Path planning algorithms enable agents to maneuver across more complex terrain where local minima would normally trap a character. DI-Guy AI creates navigational meshes for terrains that allow agents to understand the topology. No special preparation of the Open Flight terrain is required and the navigation mesh, once generated, can be saved with a scenario and used in subsequent simulations.

DI-Guy Minds: Extensible Hierarchical Finite State Machines

Agents with DI-Guy Minds are the top of the behavior pyramid. They are the smartest characters in DI-Guy. DI-Guy Minds are implemented with open Lua scripts for exceptionally fast performance and extensibility. The Minds feature an object-oriented architecture that supports inheritance, so common algorithms and behaviors are automatically shared across a family of Minds, while further specialization is as simple as a new subclass. Microthread technology allows agents to think quickly in parallel with other threads. The architecture supports a simple but powerful hierarchical finite state machine (HFSM) to control the agent's behavior at the highest level. The HFSM receives and processes messages incoming to the character from the environment, along with internal state, to determine if a high-level behavior state change should be triggered. The Mind architecture makes it easy to add new states and behavior to a Mind.



DI-Guy AI characters use Base Behaviors and Minds to determine how to move and react to ongoing events. In this picture, optional thought balloons are used to reflect the mood of the crowd. DI-Guy AI lets simulation authors instantiate intelligent crowds and then extend that intelligence.

Extending DI-Guy Minds

DI-Guy Minds can express the current state of their HFSM via any of the other authoring methods in the pyramid (e.g. – base behaviors, paths, actions, gazing, aiming, expressive faces, and gestures). For example, if an agent needs to load an artillery piece, one can add a load_artillery state to the Mind and then express it by creating a DI-Guy path that includes all the local movement and action information for where and how to perform the task. Match the best authoring technique to each desired behavior, or combine them for even more sophisticated behavior. Compelling dynamic behavior can be created by defining the various states an agent may be in and then matching the best authoring methods to each state, whether that is a base behavior, path, or direct control using action/gesture/gaze, etc. via the API. For best results, extend existing DI-Guy Minds rather than starting from scratch, so that you leverage the smarts and high performance architecture already built in.

DI-Guy AI – Getting Started

The easiest and most powerful way to author with DI-Guy AI is to use it via the DI-Guy Scenario authoring application. Blitz in hundreds of AI agents in a single brush stroke using the Crowd Blitzer. Create a variety of crowds, each composed of thinking autonomous agents, in just a few minutes. In an hour, a simulation author can interactively populate an entire city region with thousands of characters and then test/observe the complex emergent behavior that results.

To create a scenario using DI-Guy AI in DI-Guy Scenario:

- Step 1:** Select Crowd Blitzer on the Input palette and pick a Crowd Profile.
- Step 2:** Paint your crowd directly on the terrain. (As little as one click and drag of the mouse)
- Step 3:** Play the scenario.

DI-Guy AI can also be used directly from the DI-Guy SDK. To get started, examine the ai_populate programming example.

To see how to play scenarios authored in DI-Guy Scenario in a DI-Guy SDK application, examine the ai_playback programming example.

Meanwhile, use this manual and the tutorials herein to harness the full power of DI-Guy AI.

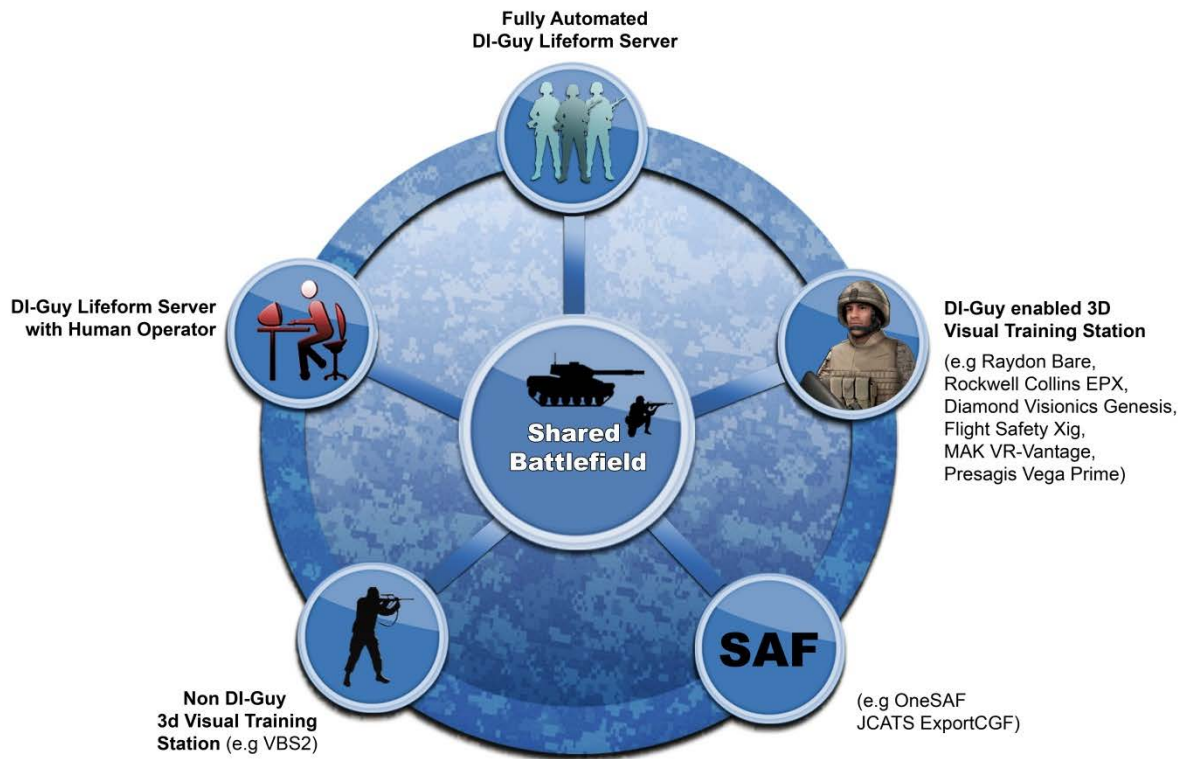
The DI-Guy Lifeform Server

Many simulations today function as parts of a larger distributed simulation using DIS or HLA networking. The DI-Guy Lifeform Server is designed to easily federate with distributed simulation environments. Just attach DI-Guy Lifeform Server to your network and populate battlefields and cities with thousands of human characters that interact with the incoming entities from other SAFs or live trainees. DI-Guy Lifeform Server can be used by itself, in

What is the DI-Guy Lifeform Server? (While the “Lifeform Server” is a powerful concept, it also can be thought of as a distinct combination of DI-Guy modules.)

DI-Guy Lifeform Server =
 DI-Guy Scenario (Visual Human Authoring Station)
 + DI-Guy AI
 + DIS or HLA Networking Module

conjunction with a human operator, or as a complement/slave to existing SAFs that model air, land, or sea-based vehicles, while performing the function it does best; *simulating realistic human activity on a wide scale.*



Typical architecture for using DI-Guy in a DIS or HLA networked exercise. A common DIS/HLA battlefield is populated by 1,2) Automated/Slave and Operator-driven Lifeform Server, 3) third party SAFs and visualized with 4,5) both DI-Guy SDK enabled and third party training stations.

DI-Guy AI works with DI-Guy SDK and supports a variety of platforms

Scenarios that use AI can be loaded into DI-Guy SDK powered applications. DI-Guy AI has a C++ API that complements and works with DI-Guy SDK. DI-Guy SDK can be integrated into virtually any computer application running on Windows 32/64 or Linux 32/64. Out of the box, DI-Guy SDK provides complete and open source rendering solutions for OpenGL, DirectX, Vega Prime, and Open Scene Graph. Best of all, the DI-Guy Graphics API lets users customize DI-Guy graphics to their specific renderer, so virtually any renderer that runs on a Windows or Linux box can use DI-Guy AI characters.

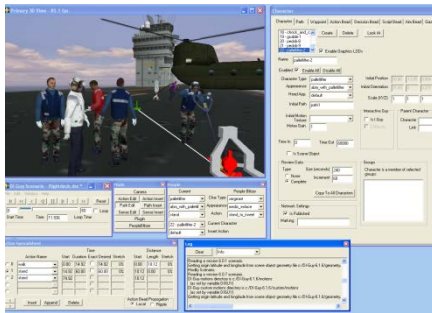
DI-Guy AI is Real Time

Everything about DI-Guy AI is real-time. Used through DI-Guy Scenario or the Lifeform Server, the user interface lets the scenario author edit crowd activity and see the people act out their roles immediately, without a compilation delay. In fact, one can manipulate crowds while the scenario plays, which is important for a real-time Lifeform Server operator.

The DI-Guy Product Line

DI-Guy AI 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.

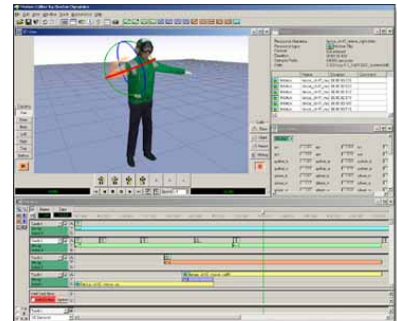


Expressive Faces Module – enable DI-Guy characters to have faces that display emotion, eyes that look in directions and blink, and lips that sync to sound files.

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.

Documentation

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

- o **DI-Guy SDK User Guide** – How to program using the DI-Guy SDK
- o **DI-Guy SDK Reference Manual** – All the functions and classes of DI-Guy SDK
- o **DI-Guy Scenario User Guide** – How to use DI-Guy Scenario
- o **DI-Guy Content Guide** – Images of DI-Guy Content and Customization Guide
- o **DI-Guy AI User Guide** – How to use DI-Guy AI
- o **DI-Guy Motion Editor User Guide** – How to import, edit, and customize DI-Guy motions
- o **DI-Guy for Vega Prime** – How to use DI-Guy with Vega Prime
- o **DI-Guy Digest** – XML-based file that explains DI-Guy content to non-DI-Guy applications
- o **DI-Guy Lua Reference** - HTML documentation of included Lua packages

Once DI-Guy is installed, 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 enables quick access to all the above documents along with additional technical information about DI-Guy.

Getting Started

Chapter 2 of this manual is a set of tutorials that covers the essential features of DI-Guy AI. The tutorials help new users become proficient DI-Guy AI users in just a couple of hours. Chapter 3 gives an in-depth explanation of the concepts that make DI-Guy AI work. Use the rest of the manual as a reference guide to DI-Guy AI. Good luck, and let's get started populating virtual worlds with the hustle and bustle of human populations!

2. DI-Guy AI Tutorials

This chapter will run you through a few tutorials that will demonstrate some basic AI scenario building techniques. To run them, start DI-Guy Scenario from the Start Menu. Consult the DI-Guy Scenario user guide for the basics on how to use that application before doing these tutorials if you are new to DI-Guy Scenario.

2.1 Tutorial 1 – Quickstart with DI-Guy AI Minds – AI World Creation

Load the scenario “AI_World_Creation.dss”. AI World Creation lets you easily work with some of our most advanced Mind characters. It includes a helpful button panel in the 3D Window to assist with populating and controlling the characters. This scenario is more than just a demonstration, you can swap out the terrain and put in your own and use it as the starting point for your scenario.



Here is a basic description of the 3D Window button panel:

Setup Terrain – replace the desert village shown in the image above with your own terrain.

Blitz Squad – create a squad of US Marines.

Blitz Enemy Squad – create a squad of enemy soldiers to fight.

Create Food Stand – place a food stand anywhere in your world. When characters get hungry, they will seek it out.

Create Mosque – place a mosque anywhere in your world. When characters have the urge to pray, they will go to the mosque.

Create Lounge – place a lounge anywhere in your world. When characters want to socialize, they will go to the lounge.

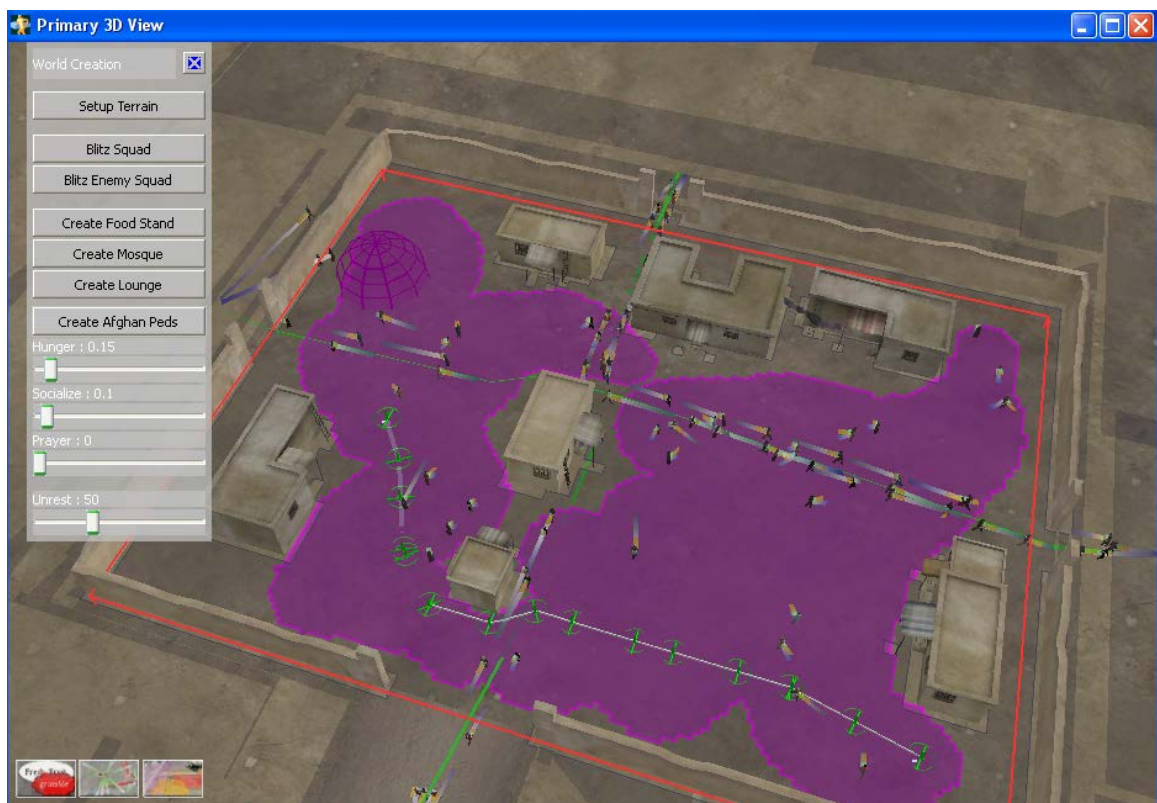
Create Afghan Peds – Create pedestrians using the base behavior travel.

Social sliders – modify the hunger, socialize, prayer, and unrest parameters of the people of the village.

Bottom Three Buttons – Toggle Thought Balloons, Toggle Vector-style graphics, Toggle region graphics.

This level of control is only the tip of the iceberg for your characters. Once the characters are instantiated in the scene, each character also has a pulldown menu you can right-click on to further specify behavior.

2.1.1 Create a civilian population with a simple pattern of life



Use the Create Afghan Peds, Food Stand, Mosque, and Lounge to add people and smart buildings to your scene. To start, create two Afghan Peds groups, one traveling back and forth between the left and right

gates, and the other traveling between the top and bottom gates. Then augment those traveling characters with some wander pedestrians. Use Crowd Blitzer in the Input Mode window to add the wander pedestrians. In the space of a few seconds you should already have populated the village with ~90 characters.

In this example, we placed the food stand in the upper right corner, the lounge down towards the left, and put the mosque outside the walls above the village.



As the simulation runs, you'll notice very quickly that the fruit stand is popular. The thought balloons let you see that some characters have interrupted their travels to get some food at the stand, saying "I'm going to eat.". As they get near the stand, they ask "Can I buy some fruit". Patrons enjoy their meal "(munch..)" and some even become happy "I'm happy" due to their full bellies. Similar behavior can be seen at the lounge and mosque.

As you learn more about the lua scripting that underlies the AI Minds, you can explore how these behaviors were created – and enhance that behavior to meet your needs.



As noted earlier, that you can command the civilians directly by right-clicking on them when in “Character” Input mode. You’ll see the following menu choices:

- o Show – shortcut to quickly access DI-Guy palettes
- o Call to Prayer – seek a mosque
- o Get hungry – seek a food stand
- o Get angry – decrease happiness
- o Get happy – increase happiness
- o Want company – seek out a lounge
- o Play dead – pretend to be unconscious
- o Plant IED – plant an Improvised Explosive Device
- o Throw Molotov – throw a Molotov cocktail at a friendly vehicle
- o Become Instigator – decrease the happiness of people the Instigator come in contact with

Take some time to play with these commands and observe the behavior.

2.1.2 Create soldiers that use formations

Select “Blitz Squad” and blitz in the squad outside the walls of the village and hit play. You’ll notice your soldiers immediately run to form a circle formation around the leader.



Note that you can command the soldiers the same way as civilians by simply right-clicking on them. You'll see the following menu choices:

- o Shoulder Weapon – Note that this may increase civilian unhappiness
- o Show – shortcut to various DI-Guy authoring palettes
- o Lower weapon
- o Take a knee – Note that this may increase civilian happiness
- o Move to point
- o Return to patrol /move to path
- o Stop AI Autoattack – Squad will react to non-friendly fire unless you turn this functionality off.
- o Formation (Column, Staggered Column, Defend, Echelon Right, Echelon Left, Line, Wedge, Vee)
- o Destroy squad – shortcut to delete this crowd.

Here are some things to try:

1. Command some new formations and watch the soldier line up.
2. Use the move to point to have your squad go somewhere.
3. Try doing a multi-point move using Move to Path. Remember, you can use the ALT key to temporarily enter Camera mode to help you lay out your path.
4. March your squad into the village. Notice how the civilians and soldiers will do collision avoidance with each other.
5. Tell your soldiers to shoulder their weapons. You should see an increase in unhappiness amongst the civilians. (Make sure you have thought bubbles toggled on to see this)
6. Now have your soldiers take a knee, you should see the unhappiness decrease and happiness occur.

2.1.3 Create an enemy squad and initiate combat

Blitz in an enemy squad inside the village. Combat should occur almost immediately between the two squads with the friendly forces eventually winning. Note how the civilians flee the scene as combat erupts.



Now try blitzing in an enemy squad outside the gates so that combat doesn't occur. If you right click on these fighters, you'll see they have the same menu choices as the friendly squad.

2.1.4 Try loading your terrain into AI World Creation

The AI World Creation scenario is designed to get you up and simulating quickly on your terrain. Simply select the setup terrain button and bring in a new terrain. It is probably best to use a fresh AI_World_Creation.dss file and make the terrain swap be your first step. The image below shows part of a squad of DI-Guys moving through the Sunnyvale database (which we swapped in in place of the default_stage_11) after being blitzed in and given a Move to Point command via the AI World Creation scenario. EXTRA CREDIT: Can you save AI_World_Creation to AI_World_Creation_Western and alter

the appearances of the civilians to be western?



2.2 Tutorial 2: Quickstart with Base Behaviors

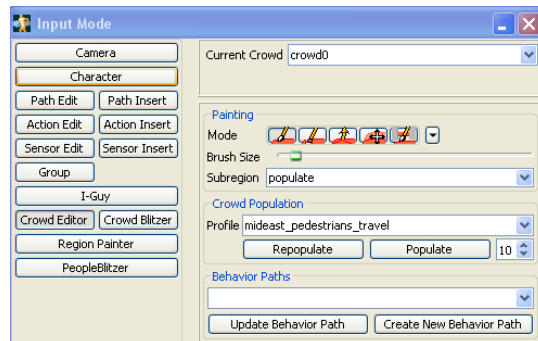
Open the scenario file `scenarios/AI_Crowd_Blitz_Demo.dss`. This will load a scenario set in the `middle_east_town` scene. In order to avoid inadvertently saving over the original scenario, immediately save the scenario with a filename such as `myCrowdBlitzDemo.dss`

You should see something like this in the Primary 3D View:

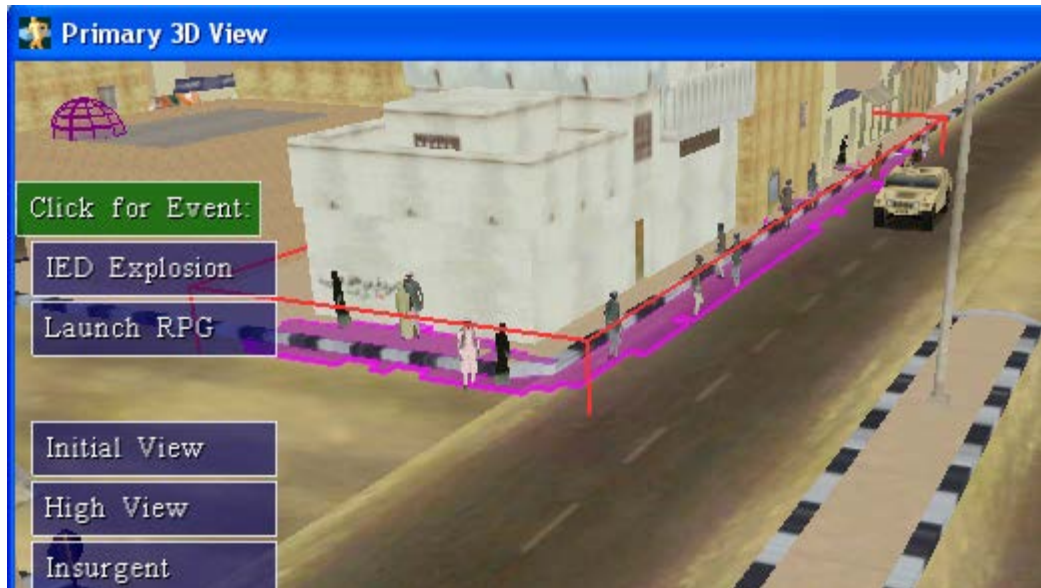


2.2.1 Create a Crowd using Base Behavior Travel

Let's jump right in and add some people to the scene. Bring up the Input Mode palette if it isn't already open by selecting the menu item Window|Input Mode, or pressing F3. Once it's visible, select the Crowd Blitzer mode, and change the Crowd Blitz Parameters Profile to "Mideast_pedestrians_travel".



Move the mouse over the 3D View and notice the purple sphere; that's the Crowd Blitz "brush". When you left-click and drag the mouse, the ground inside that sphere will be marked as the new crowd's population region. We're going to populate the upper-left sidewalk. Move the brush up the far left sidewalk for a "start position". Press the left mouse button and make one long continuous brush stroke, first down along the sidewalk to the corner, then around the corner and up the side street. When done, let go of the mouse. You should see the sidewalk become populated with 10 pedestrians.



That's it! Press Play on the Time Control palette. The pedestrians will walk back and forth, using the path shape defined by waypoints in the populate region as a general guide, and will avoid each other without any additional input from you. This walking back and forth is an example of the "travel" behavior for DI-Guy AI Agents.

Once you've got the crowd walking sidewalk, consider putting more crowds on the other sidewalks or even crossing the street.

Editing the Travel Path Shape: The travel agents use a path shape to guide their travel. You can visualize the path shape by going to the Scenario Objects palette -> Visibility tab and selecting “Current” or “All” for Path Shapes. To edit the path shape, use standard pathing techniques such as “Path Edit” and “Path Insert” from the Input Mode or find the path shape in Scenario Objects->Path Shape and try “Ground Clamp All” or edit the waypoints there directly. If you find your characters aren’t getting around the corner as nicely as you like, try pulling the corner waypoint wide.



2.2.2 Add a Wander Crowd

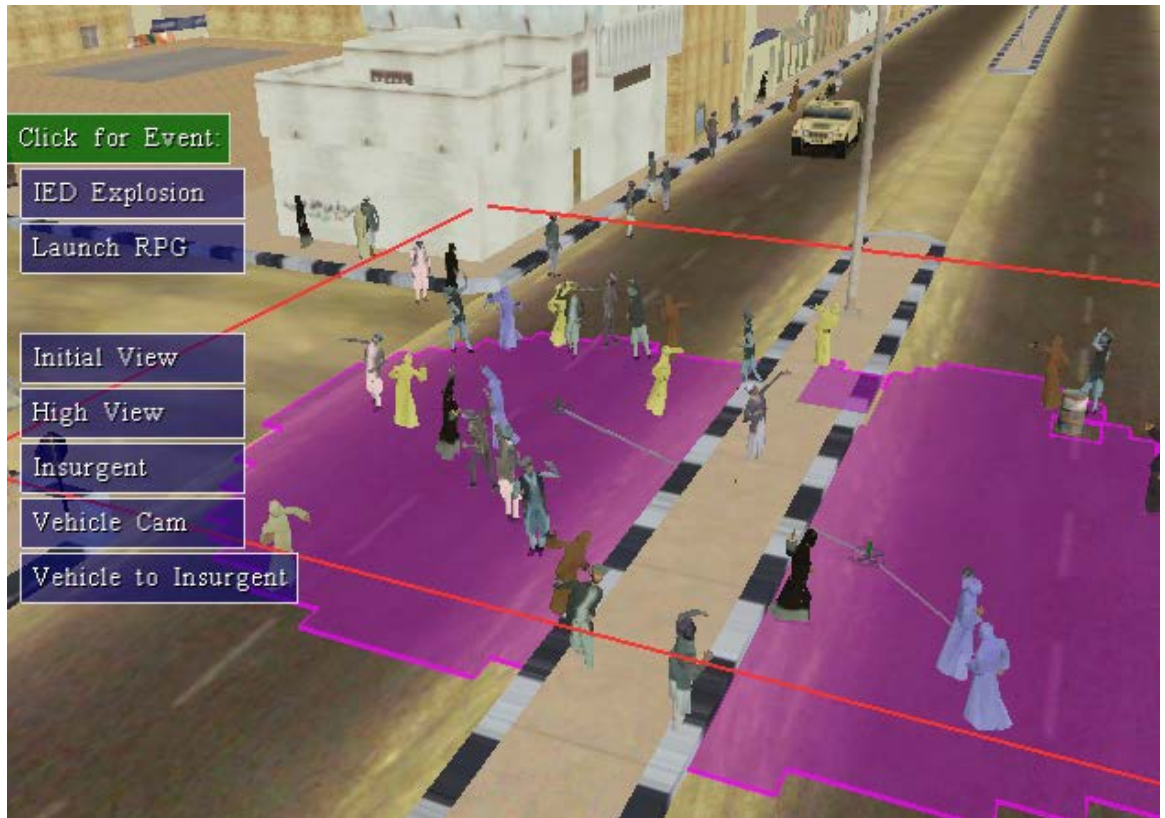
Now we’ll add some more people to the scene. This time we’ll add a wander crowd of angry pedestrians to the center of the street.

Press “Crowd Blitzer” on the Input Mode palette (it may have automatically have switched to Crowd Editor – this is an adjustable user preference). Open the Profile list and scroll down to select the “mideast_pedestrians_angry” entry. This profile adds pedestrians with middle-east appearances that will behave in an angry manner.

Now set the number of characters that will be blitzed. Change the value in the Crowd Size spin-box to 40.

Let’s also make the brush bigger. If you have a mouse with a scroll wheel, move the mouse cursor into the 3D View and roll the wheel up to increase the size of the brush. If you don’t have a scroll wheel, move the Brush Size slider in the Input Mode palette to the right.

Now that the profile and crowd size have been set, it’s time to add the new crowd. Click and drag in the 3D view to blitz in a new crowd. Paint a large area that covers both sides of the street.



This crowd uses the “wander” behavior with an “angry” variant, giving it the appearance of an angry mob. Press Play on the Time Control palette to see everyone move around. Notice that the angry crowd focuses on the humvees when waving their fist. This is because the crowd profile chosen, “mideast_pedestrians_angry” has the character “veh-0” pre-selected as its focus character. To understand this better, select the Scenario Objects->DI-Guy AI Objects->Crowd Profile palette. Here you will see the basic definitions for “mideast_pedestrian_angry”. Notice that it uses the base behavior “Wander”, the desired variant is set to “angry”, and the Focus Character is veh-0.

2.2.3 Trigger a flee base behavior

By now, you’ve noticed the blue buttons on the left side of the screen. The bottom buttons let you select different camera angles. If you start the scenario and then select “IED explosion”, you’ll notice that all the surviving characters flee the explosion area. Let’s examine how the buttons were made and how the flee was triggered.

Setting up the UI buttons – This is done in the script “initialize labels” found in the Scenario Objects->Event Handlers->Script tab. Note that the button we are interested in is “label ied”. To understand how the button calls code, switch to the Event Mapper tab and highlight the “label” type in the upper left window. If you click “modify” on the 3 callback ids in the bottom window, you’ll see they call the scripts

“pressed_red”, “explosion”, and “released_red”. The first and last scripts just handle the GUI visuals of the button. What we’re interested in is the “explosion” script. Go back to the Script tab and examine the label-style script called “explosion”. It calls two scripts, “explode_target” which handles the visuals, and “crowds_react_to_explosion”, which handles the killzone and what the survivors do.

The code for “crowds_react_to_explosion” is found in the “initialize_rpg” script. Notice how the code cycles through all the crowds, killing members within 6 meters of the explosion, and then setting the crowd paramaters to “good_flee_params”.

```
function crowds_react_to_explosion()
  local num_crowds = this_scenario:get_num_crowds();
  local cr_index;

  for cr_index = 0,num_crowds-1 do
    local crowd = this_scenario:get_crowd_at_index(cr_index);

    --
    -- Kill any characters within a certain radius of the
    -- explosion.
    -- local num_members = crowd:get_num_members();
    local ch_index;

    for ch_index = 0,num_members-1 do
      local ch = crowd:get_member_at_index(ch_index);

      if (ch:is_within_distance_n_of_character(rpg_target_name, 6.0) == 1)
then
        ch:die_now();
      end
    end

    --
    -- Load parameters from a crowd profile that cause better
    -- flee behavior.
    --
    crowd:set_current_params_from_profile("good_flee_params");

    --
    -- Flee from the exploded target.
    --
    local flee_distance = 15.0;
    crowd:agents_flee_character(rpg_target:get_name(), flee_distance);

    --
    -- Set the focus character to the exploded target so that
    -- characters will turn to face it once they've fled far
    -- enough.
    --
    crowd:set_current_focus_character(rpg_target:get_name());
  end
end
```

The “good_flee_params” can be found in the Crowd Profile tab under DI-Guy AI Objects

Crowd Profile

Ind	Name
16	nurses_wander
17	good_travel_params
18	good_wander_params
19	good_mingle_params
20	good_angry_params
21	good_flee_params
22	good_nurse_params

Create Delete
Create Copy

Name: good_flee_params

Populate Settings Default Agent Parameters

Behavior Settings Prop and Scene Object Avoidance Character Avoidance Trap Settings and Path Planning

Behavior

- ☐ None
- ☐ Travel
- ☐ Wander
- ☒ Flee
- ☐ Pursue
- ☐ Mingle
- ☐ Attack
- ☐ Idle

The Flee behavior must be triggered from decision beads or the scripting API. See diguyCharacter::agent_flee_group(), agent_flee_character() and agent_flee_crowd() as well as similar members in diguyCrowd.

Behavior Path Shape: (none)

Behavior Region: (crowd region)

Behavior Subregion: base

☐ Region Border Is Solid

Desired Posture: upright

Desired Variant: curious

Focus Character: (none)

Focus Group: (none)

Speed Settings

	Enabled	Start At Radius	Exact Action (Optional)
Still (Always)		0	(default)
Slow	☐	1.000	(default)
Medium	☐	1.000	(default)
Fast	☒	1.000	(default)
Fastest	☒	2.000	(default)

Radius Variation: 3.000

Max Orientation Change Rate: 240.000

Try modifying the scripts, perhaps increasing the kill radius, the flee distance, or creating a new “good flee parameters” profile.

2.3 Tutorial 3 – Explore other sample AI scenarios included with DI-Guy

2.3.1 AI Crosswalks

AI Crosswalks is a demonstration of Western pedestrians performing a pattern-of-life behavior in the center of a typical suburban village. This demo is a great example of how regions can enhance crowd behavior performances. Take the time to examine the regions, particularly the orange regions. A key concept in the AI Minds of the pedestrians is to pick a random point in the orange regions, navigate to it using the navmesh, and then perform a peek action before choosing another orange point to travel to. The orange regions have been strategically placed at points-of-interest in the town. The fountain, a news interview, street corners, and shop windows all have orange regions that characters will travel to, just like people in a real city.



Other regions are also important, such as sidewalks and crosswalks that are painted so that the characters will prefer these regions when doing path planning to their new destination.

2.3.2 AI Schedule

☒	12:00	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get hungry	Percentage: 20
☒	12:01	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: want company	Percentage: 30
☒	12:02	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get angry	Percentage: 20
☒	12:03	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: call to prayer	Percentage: 30
☒	12:04	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get happy	Percentage: 40
☒	12:05	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get angry	Percentage: 10
☒	12:06	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get hungry	Percentage: 20
☒	12:07	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: want company	Percentage: 30
☒	12:08	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: get happy	Percentage: 20
☒	12:09	Action: Trigger Signal	Role: luaMiddleEasternPedestrian	Action: call to prayer	Percentage: 30

The scheduler is a new feature of DI-Guy AI. It assists with setting up patterns-of-life behavior by letting the author schedule the daily routine of the characters. The demo compresses time for demonstration purposes. As you can see in the screen grab of the scheduler (from the DI-Guy AI Objects->Schedule Editor palette) at 12:00 a signal is triggered to all members of the Group (the Group selection is not pictured but lets you choose “all”, any of the crowds in your scenario, as well as “net_friendly”, “net_neutral”, “net_opposing” or “vehicles”) whose Mind hierarchy is a subclass of

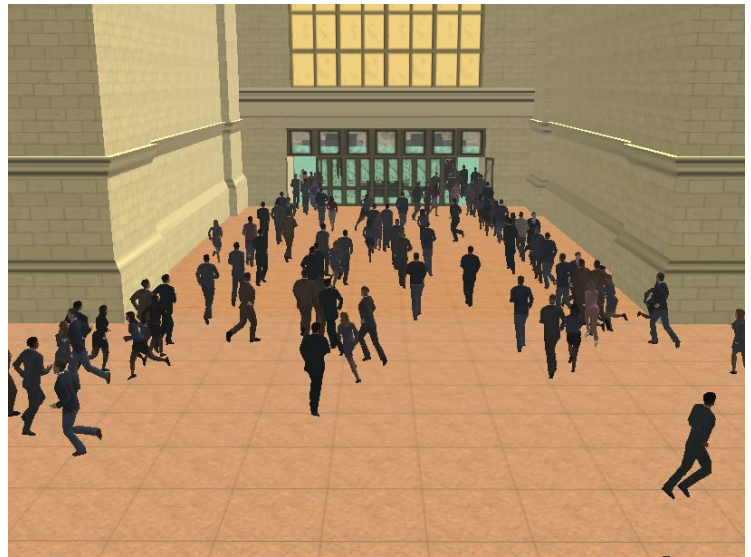
“luaMiddleEasternPedestrian to “get hungry”. When you run the demo you will see 20% of the inhabitants get hungry and seek out the food stand. At 12:01, 30% of the inhabitants are triggered to “want company” and seek out the lounge. Imagine creating a town where the kids go to school in the morning, the adults head off to work, the elderly go to the park, people go to restaurants at lunchtime, etc. This is now possible for the author to create using the AI Schedule Editor.

Note that all characters do not all receive a message at exactly 12:01 to “want company”, instead the scheduler spreads out this communication to the target agents across a number of minutes specified as the “Time Window” (Time window selector not pictured) to make the execution of the schedule seem natural and not robotic.

2.3.3 AI Train Station Alarm

The AI Train Station Alarm is a demonstration scenario with concepts applicable for emergency response and other conditions where an evacuation simulation is desired. The scene starts with commuters at a large urban train station whose normal routine is interrupted when a fire drill occurs. Characters then head towards the nearest exit, making an orderly exit from the building.

This demo takes advantages of the concept of “Region Border is Solid”, a checkbox in the crowd Behavior Settings tab that doesn’t let characters navigate outside of the



behavior subregion. A simple script sets up DI-Guy triggers for setting the alarm and signaling whether exit doors are closed are not. When the alarm is triggered and the exit doors open, the agents wander to the red exit regions (these can be difficult to see since they overlap the green regions), and do their wander parameters, will jog or otherwise move quickly to reach the red area.



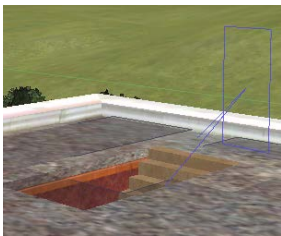
2.3.4 AI Multi-Floor Path Planning

This demo shows how to setup your regions so that agents can navigate through multi-floor buildings.

First, play the scenario. You'll see three agents be commanded to go to the target oil drums that are 2 stories away from the agent.

Examine the Regions tab. Here you will see, in addition to the ground level navmesh, 4 other regions defining a 2nd floor and roof for two different buildings. Note that landings half way between the floors can be assigned to these regions, as long as they are not overlapping another part of those regions.

Next, examine the Path Shapes tabs. Here you will find some special path shapes. There is one for each stairway (4 in one building, two in the other). Each path shape can only have two waypoints. Notice that the checkbox "Use as Navmesh Portal" has been selected. If you go the Visibility Tab and select "Regions All", you will see a special portal graphic at the end of a path shape.



Finally, examine script "Script0" in the Script tab. For each pedestrian, the x,y,z position of an associated target oil drum is retrieved and used as an argument to an agent_move_to_point() function call, along with the wildcard "*" to designate that any combinations of regions can be used. The roof regions are passed as the behavior region in the agent's parameters. "Script0" is executed at time 1.

In summary, if you define separate regions for each floor of a building (note you will want to use the new region preferences "Z sphere above (roofs)" and "No sphere test (stairs)" when painting your regions) and create path shapes for each stairway with portals, you will be able to add multiple levels to your worlds.

Let's see if we can duplicate the above starting with new.dss for a single character.

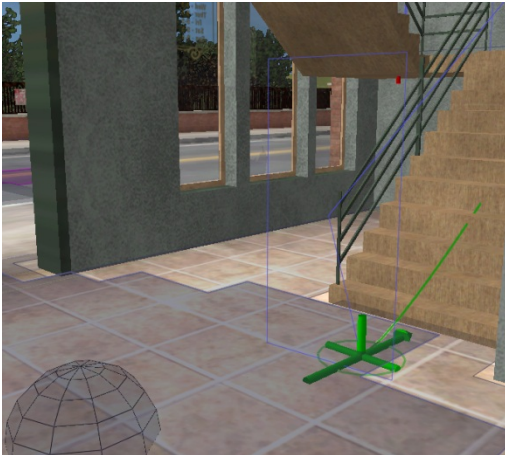
1. Load new.dss. Save as a "myMultifloor.dss" so that you don't overwrite new.dss.
2. Crowd Blitz a single character onto the road in front of the building with the glass-windowed stairs. Use "pedestrians_wander" for the profile. Rename your character to "multi-floor-ped" via the Character Objects->Character tab.
3. PeopleBlitz a barrel (character_type: prop, appearance: drum_rust) onto the road nearby. Name the barrel "target".
4. Cut and paste the following script code into a new script "MoveToTarget" in the Event Handlers script tab. Make it be a Timed event that triggers at Time 0.



```
local character_target = this_scenario:find_character("target");
local character = this_scenario:find_character("multi-floor-ped");
local res,tx,ty,tz = character_target:get_position();
character:agent_move_to_point(tx,ty,tz,"");
```

Note: the above script uses the `agent_move_to_point()` method which can cause high CPU usage and therefore noticeable frame-rate irregularities should the path required be long or difficult to compute, especially if multiple characters perform the same script at roughly the same time. As an alternative, there is a threaded background version of this planning method (called `agent_move_to_point_bg()`) that is described in a later section (Section 3.4).

5. Run a quick test. Verify that the character moves to the target.



6. Go inside the building to the ground floor. On the Input Mode select “Create Region”. Name the region “groundfloor”. Paint the ground floor space at the bottom of the stairs.

7. Goto the 2nd floor. On the Input Mode palette, use the Mode Preferences pulldown to select “No Z Sphere Test (stairs)” for your region painting mode. Paint the landing between the 1st and 2nd floors. Paint the second floor up to the lip of the stairs from below. Name the region “secondfloor”.

8. Go to the Character page for the target and deselect “Is Scene Object”.

9. Add the following code to the script:

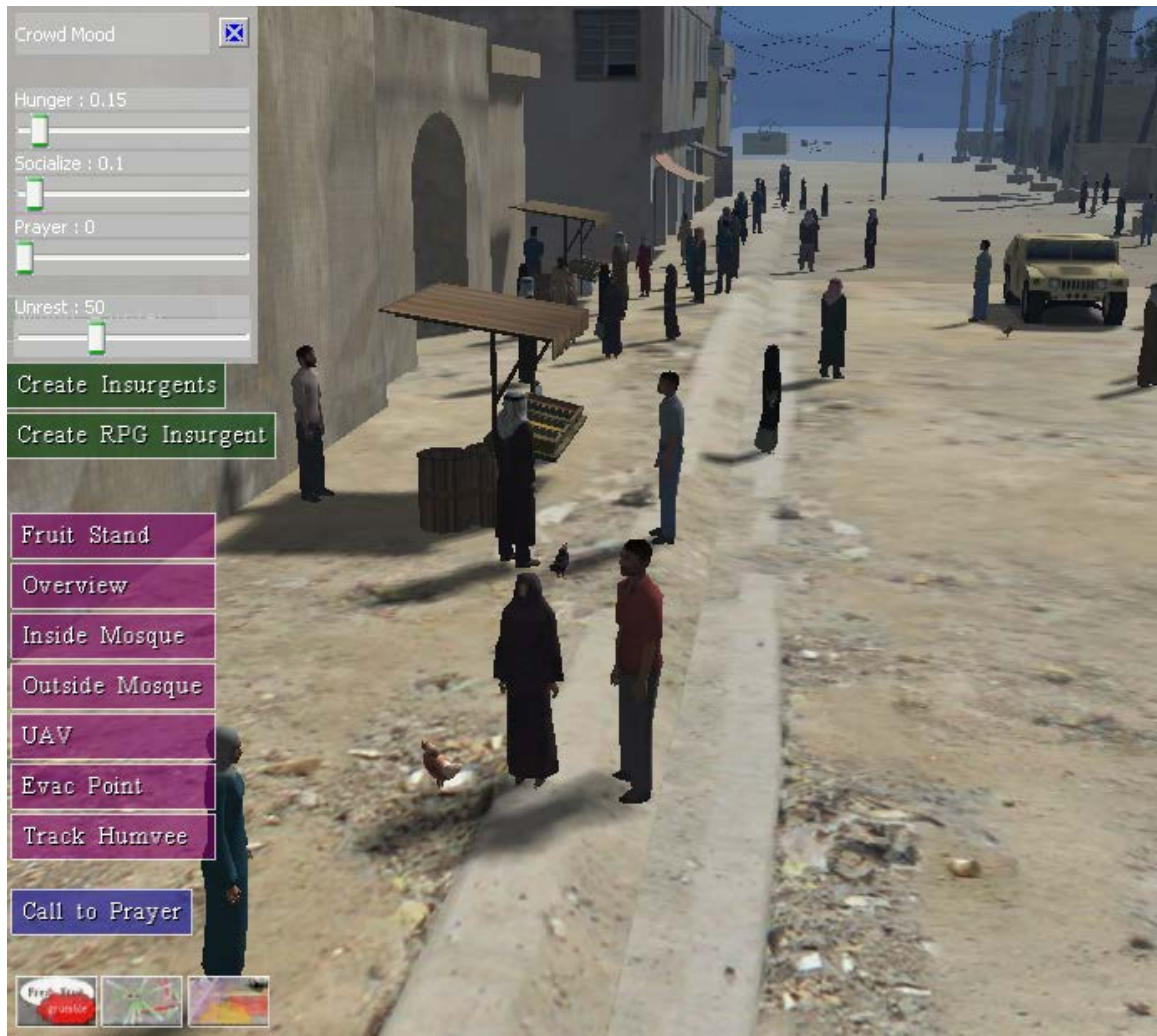
```
local params = character:agent_get_current_params();
params:set_behavior_region("secondfloor");
```

10. Create two path shapes in the Scenario Objects->Path Shape tab. Create two waypoints for each path shape. You’ll need to now position those waypoints. One way is to use the pulldown menu Edit->Preferences for overall DI-Guy Scenario control and select “Show Cursor Position” in the “General Settings Tab”, then position the mouse where you want the endpoints of the path shapes to be (the first path shape will go from the 1st floor to the 2nd floor landing, and the second from the 2nd floor landing to the 2nd floor) and type the x,y,z coordinates into the waypoints. Once you have that, you should use “Path Edit” mode to move the waypoint to the exact correct position (will also ground clamp the waypoint better) and shorten the weights of the waypoint.
11. Don’t forget to select “Use as Navmesh Portal” for each Path Shape.



12. Run the scenario. You should see the character move up the stairs to the target. To debug, be sure to use the Visibility tab to visualize the portals and the regions.

2.3.5 AI Middle East Town



This demo contains many of the features already seen in the AI World Create and the AI Crowd Blitz demo. Unlike those two demos which focus on populating, this world is already fleshed out. There are some interesting features in this scenario. The user is encouraged to explore the scripts in the scenario to see how the advanced functionality was implemented, and to use those ideas to implement new advanced behaviors.

Animals – note how this demo leverages DI-Guy animal content to increase realism.

Sounds – like animals, sounds really help increase immersion.

Humvees – this demo has DI-Guy controlling the vehicles, but is inspired by the idea that this world would be what trainees in a humvee/convoy trainer would see.

Call to Prayer – Select the call to prayer button, watch how the crowd responds to this message broadcast from the Mosque and make their way to the building. Once in the building, the characters segregate to different sides of the partition based on their sex. Upon arrival in the building, the characters engage in prayer. This entire behavior is an excellent example of extending minds and the states of the FSM to enable complex behavior started by an environmental interrupt.

IED Placement – Right click on one of the pedestrians (remember, you need to be in Character Mode in the Input Mode palette to access these individual pulldowns) and select “Plant IED”. You’ll be prompted to select an emplacement point. Select a point in the road you know the humvees will drive over. You’ll see the pedestrian plant the IED, and later, the humvees running over the IED resulting in an explosion.

Deploy Troops – Right click on a humvees and disembark troops.

Create RPG Insurgent – Find the curved archway just past the fruit stand on the left. Select the 3D Window button “Create RPG Insurgent”, place him against the far wall, and select “Wait for Target” when prompted. This demo shows two interesting features: 1) The ability to do Line-of-Sight tests to trigger behavior, and 2) How to coordinate the aim, creation of ballistic, smoke special effects, and intersecting explosion with the target.

Retreat/Evac Point – Disembark troops from a humvees and select Retreat/Evac Point. There is a designated evacuation area at the far end of the town. Again, a coordinated simulation is demonstrated that has the soldiers move to the evac point. Upon arrival, the circling helicopter lands and embarks the troops, and then lifts off.

2.4 Tutorial 4: Creating a AI subclass (package) from scratch

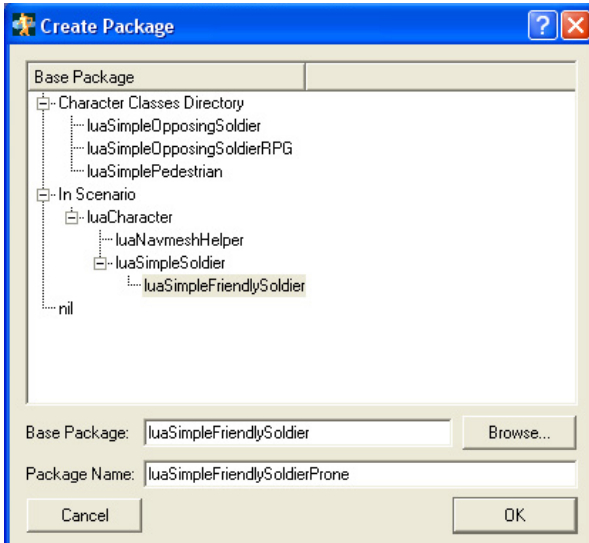
In DI-Guy AI, a package is a Lua script, either on disk (in the \$DIGUY/config/diguy/scripts/<subdirectory> folders) or coded directly in a scenario file script, that is automatically compiled at scenario load or whenever the script changes, and supports a notion of dependencies. Typically, packages contain code that is run at initialization as well as code that executes at runtime. Packages are used to define new subclasses for extending the behavior of DI-Guy AI Minds.

In this tutorial, we are going to create a new package “luaSimpleFriendlySoldierProne” that extends the behavior of a luaSimpleFriendlySoldier – the new subclass will have the ability to order the soldier to go prone, in addition to all the abilities he will inherit from the luaSimpleFriendlySoldier base class.

1. On the Lua Mind Editor tab of the DI-Guy AI Objects page, select “Create”.
2. You now have a choice between
 - Character Mind – use this only when you want to add a DI-Guy AI Mind to a previously created character who doesn’t have one. Note that the mind will be added to the currently selected character.

- All Crowd Members – use this only when you want to add a DI-Guy AI Mind to all the members of a previously created crowd. Note that the mind will be added to the currently selected crowd.
- Scenario Package – use this when you want to create an entirely new package (subclass).

3. Select Scenario Package
4. You now have a choice between New Package and External File.
5. Select New Package
6. We now need to designate the base package (base class) for our new soldier class. Select luaSimpleFriendlySoldier, since we want to get all the existing functionality, and then add the prone ability. For the Package Name, enter luaSimpleFriendlySoldierProne. The new package is initialized with some automatically generated code.



Before we add new functionality to the new package, let's quickly create a new crowd profile that will use the new mind for test purposes, and blitz in some soldiers.

7. Go to the Crowd Profile tab and select soldier_travel_lua_mind from the existing Crowd Profiles. Select Create Copy. Rename the new copy soldier_travel_lua_mind_prone. Then, in the Mind Class pull-downs, change the luaSimpleFriendlySoldier entries to be luaSimpleFriendlySoldierProne. If this option is not visible in the pull-down list, go back to the mind editor and compile the prone package you added.

8. Blitz in two sets of soldiers, one using soldier_travel_lua_mind and one using

soldier_travel_lua_mind_prone. This way we can compare our new "prone" soldiers to the pre-existing soldiers.

9. Hit "play" and verify that all the soldiers are moving.

To make our coding easier, let's make it that soldiers can only go prone if they are first commanded to "take a knee". To make the "go prone" selection available in the pulldown menu, let's override the existing knee_state() function in luaSimpleSoldier with a new knee_state() function in luaSimpleFriendlySoldierProne.

10. Copy the knee_state code from luaSimpleSoldier and place it in luaSimpleFriendlySoldierProne.
11. Make some minor modifications 1) change the function declaration to use luaSimpleFriendlySoldierProne and 2) add "go prone" to the list of UI signals:

```
function luaSimpleFriendlySoldierProne:knee_state()
```

```

self:begin_state("knee_state");

self.ui_signals = "shoulder weapon, stand, go prone, move to point, return to
patrol";
.
.
.

```

12. Because the scripts are compiled on-demand, we can test this immediately. Select “reset” and then “play”. Right-click on the character (remember you need to be in “character” input mode) and command the character to take a knee. Right click again and you should now see “go prone” now as an option in the GUI. Don’t select it yet...we need to fill out the state.

13. Now let’s create a brand new prone state, based entirely on the knee state.

```

-----
function luaSimpleFriendlySoldierProne:prone_state()
    self:begin_state("prone_state");

    self.ui_signals = "shoulder weapon, take a knee, move to point, return to
patrol";

    local params = self.character:agent_get_current_params();
    self.character:agent_begin_behavior("idle");
    params:set_variant(DIGUY_MOTION_VARIANT_NORMAL);
    params:set_posture(DIGUY_MOTION_POSTURE_PRONE);
    while 1 do
        local ret = self:process_message(self:sleep(1000000));
        if(ret) then
            return ret;
        end
    end
end

```

Note that we edited the begin_state argument, switched “take a knee” with “stand” in the ui_signals, and command the posture to be prone.

14. The last step is to edit our message processing, so that the signal “go prone” (triggered via the GUI – note that the signal has the same name as the gui entry) is recognized and acted on. The base class function luaSimpleSoldier::process_signals is the place to start, as it is here that “take a knee” is processed. Copy portions of the luaSimpleSoldier::process_signals functions to create a new luaSimpleFriendlySoldierProne::process_signals() functions. This function will only need to process the “go prone” message and will then pass control to the base class:

```

function luaSimpleFriendlySoldierProne:process_signals(message_object)
    local message = message_object["message"];

    if(message == "go prone") then
        return self:prone_state();
    end

```

```
return luaSimpleFriendlySoldier.process_signals(self, message_object);
end
```

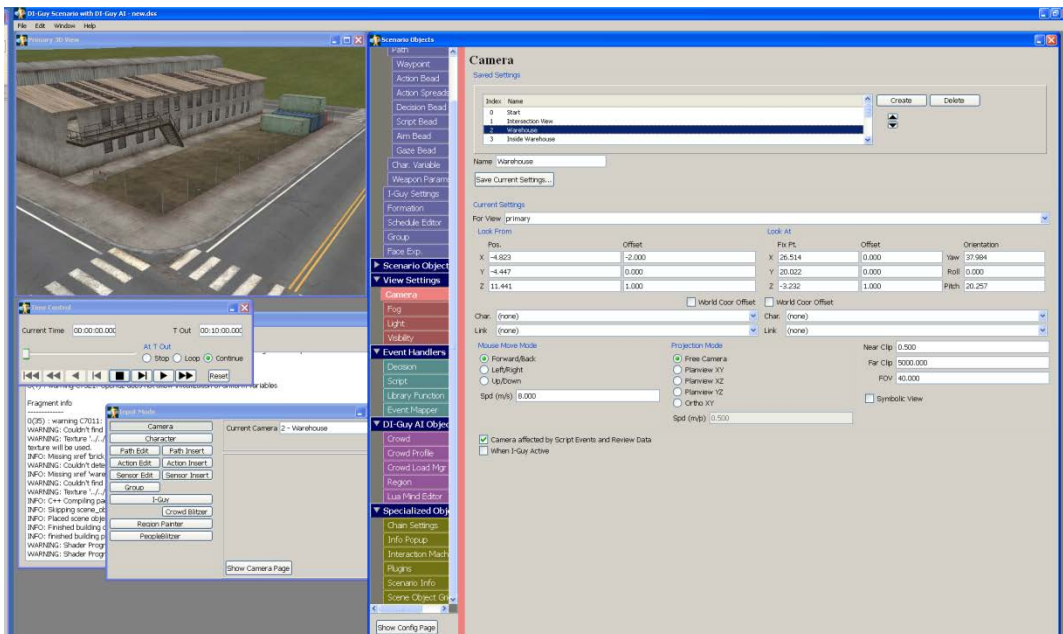
Note that if the incoming message is not “go prone”, the parent process_signals() is called and continues with all the normal processing. Be careful with the syntax here, you need to use a ‘.’ and not a ‘:’ between luaSimpleFriendlySoldier and process_signals, and you also need to pass in “self” as a lead argument.

15. Test your code by commanding your character to “take a knee”, then tell him to “go prone”, and then back to “take a knee”. If your character does not do the right thing, make sure you are checking the Log Window in DI-Guy Scenario for helpful debug information.

2.5 Tutorial 5: From Character to Agent, Step by Step

This tutorial will take single DI-Guy character and turn him into an agent. At first, we'll use a base behavior to make the agent wander in a region that we create. Then, we'll write a Lua script to move the character to a random point in the region. Then, we'll turn that script into a Lua Mind.

To start, launch DI-Guy Scenario. We are going to use the default scene geometry. Make sure the Scenario Objects window is open and switch to the Camera page. Switch to the Warehouse camera (shortcut: hit “3” when 3D Window has the mouse focus). You should see something like this:

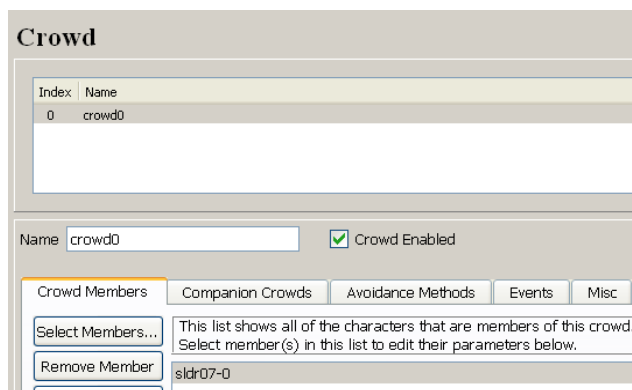
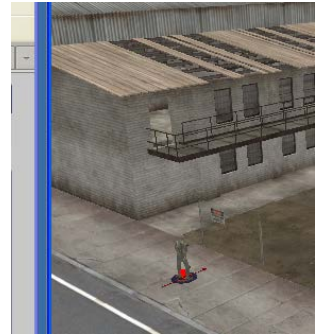


Use the PeopleBlitzer to add a single character to the sidewalk. You can use the default PeopleBlitzer settings.

Character to Agent

This character is not an agent. To turn him into an agent, open the DI-Guy AI Objects->Crowd page of the Scenario Objects window and click the Create button. This should create an empty crowd, crowd0, which will be displayed in the listbox.

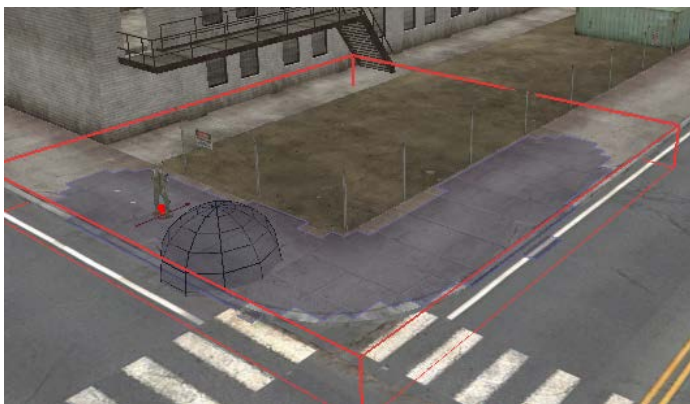
To add the character to crowd0, click "Select Members..." in the Crowd Members tab. In the "Select Character" dialogue box, highlight the character and select OK.



Creating a Region

We want the character to wander in the sidewalk. We're going to use an AI base behavior to do that, but first we need to create a region that defines where the character should walk. Regions are key to many AI behaviors. They can be created through the UI or through script. You can also access them via script when extending behaviors.

Painter mode in the Input Mode window. Select Create Region. The region will be called region0 by default. When you mouse over the Primary 3D View, you should see a hemisphere that indicates the extent of a "paintbrush" that will create a region on the terrain. You can use the middle mouse button to change the size of the brush. Use the brush to paint a region over the sidewalk. If you mess up, you can erase part of the region using the eraser mode (shortcut hint: use the ctrl key to activate the eraser).



Specifying the Base Behavior

Now, we're ready to have the character wander in the region we just painted. Open the Crowd view in the Scenario Objects window and select crowd0. At the bottom of the page is a section called Agent Parameters. This is where you can specify base behavior parameters to members of the crowd.

Change the behavior from None to Wander. Set the Behavior Region to region0, the region we just painted. Let's make the character wander about while aiming. To do this, change the Desired Variant drop down to "aim". In the Wander/Mingle settings box, change the Move On Interval Min to 1 and the Max to 5 seconds. This

means that the character will wait anywhere between one to five seconds before moving on to the next destination. Change the Max Distance to Wander field to 16 units.

Hit play. Your character should wander about region0 randomly while aiming. Feel free to experiment with the base behavior parameters. Note that our crowd only has one agent but everything you've learned applies to multiple agent crowds as well.

You may notice that your character may run on occasion when his next stopping point is far away from his starting point. This is due to a property of most base behaviors known as speed zones. Characters far from their goal will move faster than characters near their goal. Controlling these settings is described in detail later in the guide.

Let's illustrate an important point by making a slight modification to our example. Open the "Trap Settings and Path Planning" tab in the Agent Parameters section. Uncheck the field "Wander Uses Path Planner" and play your scenario again. If you've painted the region as we've shown in the picture such that the region wraps around the fence, the character will eventually get stuck behind the fence. This is because, much like a mouse in a maze, the agent is solely using local information to navigate. With the path planner enabled, an A* search algorithm is used to search some subset of the region to find a path to the goal.

Mimicking the Wander base behavior via Scripting

If you're comfortable with scripting, let's see how we can control the character's behavior via a Lua script. Set the character's base behavior back to "None" in the Agent parameters section.

Open up the Scenario Objects->Script tab. Switch the language to Lua. Click "Create" to create a new script. By default, this will create a script that triggers, or runs, when the scenario clock reaches one second. This default is fine for now. Enter the following Lua code:

```
-- get a handle to the region we painted
-- note that this_scenario is a global variable
local region = this_scenario:find_region("region0");

-- get a random point in the base subregion
local res, x,y,z = region:get_point_in_region(DIGUY_SUBREGION_BASE);
-- print the point to the log window
print(x,y,z);

-- get a handle to the agent. note that prefixing the variable name
-- with character allows tab completion in the editor to work
local character_slidr = this_scenario:find_character("slidr07-0");

-- agent_move_to_point is an api call that does what you might
-- expect
character_slidr:agent_move_to_point(x,y,z);
```

Note: the above script uses the `agent_move_to_point()` method which can cause high CPU usage and therefore noticeable frame-rate irregularities should the path required be long or difficult to compute, especially if multiple characters perform the same script at roughly the same time. As an alternative, there

is a threaded background version of this planning method (called `agent_move_to_point_bg()`) that is described in a later section (Section 3.4).

If you play the scenario now, the character should move to the point that is printed in the Log window once a second has elapsed. You can also hit the trigger now button on the script page to run the script again. Effectively, you are mimicking the Wander Base Behavior.

Create a subclass Mind specific to your Agent

Change the script's Event Type to Manual so that it will no longer run (unless somebody presses the Trigger Now button). Check that by playing the scenario. The agent should not move. We're instead going to have the script expressed as a new state.

First, let's add a mind to our agent. Switch to the Lua Mind Editor page and click the Create button. You'll be prompted to select the code block type. Select Character Mind. This mind will be a subclass named `sldr07_0`. When prompted for the base class to use, select Already Loaded.

The easiest way to get started with AI Minds is to extend an existing Mind. Lua supports object-oriented programming and we're going to take advantage of that here by modifying the behavior of an existing Mind, `luaSimpleFriendlySoldier`. Select it from the list of potential base classes in the dialog.

A script for the class `sldr07_0` should now appear in the Lua Mind Editor that is a skeleton for the Mind you are going to create. On the left side of the editor is a tree view of the objects that make up this mind. You'll find the object we are going to edit, `sldr07_0`, but there's also the objects it inherits from, `luaSimpleFriendlySoldier`, `luaSimpleSoldier`, and `luaCharacter`. Documentation for these classes can be found in the Lua html documentation included with DI-Guy AI.

If you explore the 4 classes, you'll see that our character already has a pretty complex mind. Try and ignore that for now as we're going to override a few methods to allow us to create a simplified agent whose pulldown will trigger our single pass wander behavior.

Add trigger to the GUI pulldown for your Agent

The first method we'll change is `get_ui_signals`. Currently, there is a commented out shell function ready for you to work with in your `sldr07_0` subclass. It should look like this:

```
-- This function returns a comma separated list of signals that the agent
-- is currently willing to accept.
--function sldr07_0:get_ui_signals()
-- return self.ui_signals;
--end
```

It's commented out meaning that our Mind handles whatever signals currently handled by `luaSimpleFriendlySoldier`. This method is used by DI-Guy Scenario to create a context menu that appears whenever you right click on an agent in the 3D viewer. Clicking on the item sends a signal to the agent with the corresponding label. Uncomment the method and change it to return the string "trigger":

```
-- This function returns a comma separated list of signals that the agent
```

```
-- is currently willing to accept.
function sldr07_0:get_ui_signals()
    return "trigger";
end
```



Make sure you select Character mode in the Input Mode window and play the scenario. When you right click the agent, trigger should show up as an item in the context menu. Clicking on it will have no effect, in fact the log will show a warning that there is no handler.

In order for path planning to work, Minds require a special region to be created in your scenario called navmesh. The new scenario comes with a navmesh. See Tutorial 6 for information on creating a navmesh.

Ok our Mind now has a state that waits indefinitely for the user to click the trigger option in an agent's context menu. Then, it transitions to my_wander_state which doesn't exist yet either. Add the following code to your Mind in the editor:

```
-----
-
-- My Wander State is called when "trigger" is activated. It will find a new
-- point in region0 and move to it, and then return to patrol state.
function sldr07_0:my_wander_state()
    local state_info = self:begin_state("my_wander_state");

    local region = this_scenario:find_region("region0");

    -- get a random point in the base subregion and print it
    local res, x,y,z = region:get_point_in_region(DIGUY_SUBREGION_BASE);
    print(x,y,z); -- nice for debugging - but comment this out later

    -- call simple move to state and then go back on patrol.
    self:simple_move_to_state(x,y,z,nil,nil,2,1);
    return self:patrol_state();
```

Note that when coding, certain naming conventions enables “smart browser” UI features of the Lua Mind Editor like text completion, function prompts, and argument lists. The conventions are described in detail later, but when creating a new state, choose a function name that ends with the suffix “_state”.

Save your scenario and run it. Your agent should wait patiently for you to click on the trigger signal in the context menu. Once you do, he'll move to a new point in region0, much like the Wander base behavior. Hopefully, this highlights the power of the Mind system DI-Guy AI delivers. This wander state is not as configurable as the base behavior but you can easily make it so by expanding your simple script. You are limited solely by your imagination!

2.6 Tutorial 6: Create a navmesh

The “new” scenario comes complete with a navmesh. If you are working with a terrain that does not have a navmesh, you’ll need to create one for intelligent path planning. To do so, click the Reset button and go to the Region page of the Scenario Objects window. Delete the current navmesh. Click the Generate Navmesh button (NOTE: do NOT create a region and call it navmesh) and click Ok to accept the default options. Creating a global navmesh is an expensive operation so expect this operation to take a while.

To test, create a new scenario, then replace the Scenario Objects ->Scene Object scene_object1 that is the default stage with sets/neighborhood/flt/neighborhood_base.flt. Then follow the directions from above to generate a new navmesh for the neighborhood. Test it by crowd blitzing some soldiers_travel_lua_mind agents, select character mode, and right-click on a soldier. Then have move somewhere blocked by a wall or house.

3. DI-Guy AI Concepts

DI-Guy AI's primary mission is to enable quick population of scenes with large groups of characters. It automates the creation of hundreds of characters that can sense things in their environment and have a good range of built-in behaviors with which to react to their surroundings. The author isn't overly burdened with authoring hundreds of individual performances. Agents are DI-Guy characters controlled via the autonomous crowd AI described in this chapter (as opposed to more classic methods of controlling DI-Guy characters either via direct commands to the API or by following paths). A group of Agents is called a Crowd.

3.1 Agents and Crowds

Agents

An Agent is a DI-Guy character who uses AI to determine his actions. These AI capabilities build on the standard assets that already come with any DI-Guy character (a diverse array of actions and appearances, the ability to follow paths, a robust API, etc).

Agents can:

- Sense obstacles in their environment.
- Modify their actions based on their current objectives and the sensed obstacles.
- Participate in Crowds of large numbers of other Agents.

Crowds

A Crowd is a group of Agents. Agents in a crowd are aware of each other and their surroundings. They by default interact with each other, and may share certain behaviors and goals. Based on the parameterization of the agents in a crowd, interesting and autonomous behaviors emerge.

Crowds containing many agents are created quickly in DI-Guy Scenario by painting directly into the 3D scene using the Crowd Blitzer brush. A Crowd Profile is first selected that specifies the number, character types, and appearances of the agents that will be in the crowd. Crowd Blitzing should be performed with a single click, drag, and release of the mouse; the resultant painted region is immediately populated by the specified number of agents who are ready to simulate.

A Crowd in DI-Guy AI:

- Contains zero or more Agents
- Has one or more modifiable Regions in the scene in which the Agents will operate and optionally be confined
- Specifies the method the agents in the crowd use to detect obstacles in the scene
- Has a list of Companion Crowds with which it can interact
- Has member functions that command and change parameters of all or some of the agents in the crowd at the same time

NOTE: An Agent can be part of only one crowd.

3.2 Crowd Profiles

A Crowd Profile is a description of the range of appearances and behaviors for the agents that will be instantiated when creating a crowd. A well-composed Crowd Profile makes populating scenes easy, since you can quickly create very large crowds with diverse appearances and compelling emergent behavior already built-in and ready to go.

The screenshot shows the 'Scenario Objects' window with the 'Crowd Profile' tab selected. The 'Crowd Profile' window is open, displaying a list of crowd profiles and a 'Populate Entries' table.

Crowd Profile List:

Index	Name	Library
35	afghan_pedestrians_mingle_lua_mind	builtin_crowd_profiles
36	afghan_pedestrians_angry	builtin_crowd_profiles
37	chickens_wander	builtin_crowd_profiles
38	horses_wander	builtin_crowd_profiles
39	good_travel_params	builtin_crowd_profiles
40	good_wander_params	builtin_crowd_profiles

Populate Entries Table:

	Type	Appearance	Proportion	Mind Base Class	Mind arg1
1	chicken	chicken_1	1.0	(none)	
2	chicken	chicken_2	1.0	(none)	
3	chicken	chicken_3	1.0	(none)	

Entry Selection Method:

☒ Random by Proportion
☐ Sequential

Default Crowd Size: 20

The Population Entries Table of the Crowd Profile contains a description of all the agents that will be used to populate a new crowd. It contains one or more entries, each of which specifies:

- A DI-Guy character type
- A DI-Guy character appearance

- A proportion

All DI-Guy character types are valid as entries in the Population Entries Table. Remember that in DI-Guy, a character type defines the set of actions (walk, run, jog, etc) a DI-Guy character can perform, as well as the set of visual appearances that character can have.

The appearance is the visual representation of the character (e.g. – tall man in a black suit, short man in a brown suit, etc), and is only limited to the set of appearances available for the chosen character type. See the DI-Guy User Guide for information on customizing your own appearances and actions.

The proportion field specifies how often characters with the particular entry's type and appearance should occur in the crowd relative to other entries' proportions. For example, if entry 1 has a proportion of 10, and entry 2 has a proportion of 20, characters with entry 2's character type and appearance will occur twice as often as those of character 1's. This allows different ratios of characters to be created when generating a crowd using the given profile. For example, in most groups of pedestrians, the relative number of people who use wheelchairs is smaller than the walking public. In this case, the proportion of the wheelchair entry should be set small relative to other entries' proportions.

In addition to the Population Entries, a Crowd Profile also specifies a default crowd size and default parameters for behavior, obstacle avoidance, etc, that each Agent in the Crowd created from the profile will have.

3.3 Agents and Behaviors

Crowd Behaviors and their parameterizations describe how crowds navigate around the scene.

Each Agent has a large set of parameters that define its behavior including:

- How the Agent should detect and avoid static obstacles
- How the Agent should detect and avoid dynamic obstacles such as other agents in its Crowd as well as agents in Companion Crowds
- The Base Behavior of the Agent
- The desired Posture and Variant the Agent should use when selecting actions.
 - *Postures* define the basic posture of the action: upright, crouched, and prone are the three options.
 - *Variants* define an abstract emotional and/or mental state of the character. Along with posture, the variant is used by DI-Guy AI to query the existing actions the character can perform, and then select the best or closest match when performing a still or moving action. Examples of variants include “angry” and “injured”.
- The path shape and/or region the Agent should use to guide its behavior
- An optional focus character or group of the Agent's behavior
- “Untrap” settings that specify how the Agent should attempt to remove itself from situations in which it becomes stuck or stops making progress toward where it wants to be



The high-level objectives of an Agent are selected by setting the Agent's *behavior*. Behaviors tell the Agent what it should try to be doing, though other factors in the environment such as obstacles and dangers may interrupt or modify the behavior.

Depending on the particular behavior chosen, an Agent's behavior path shape or behavior region may affect the overall behavior. These effects will be described for each base behavior below.

Similarly, the Agent's focus character and focus group may also affect the overall behavior.

When a behavior has a focus group, typically the behavior will select one character from the focus group and make it the temporary focus character. Therefore, whether a focus character or focus group is being used there is commonly only a single character at any one time that is being focused on. This character is called the current focus character.

It is easy to change the "flavor" of behaviors by changing the desired posture and desired variant of the behavior. Changing the variant from "normal" to "angry", for example, can quickly change the actions the agents in a crowd use from looking peaceful to looking aggravated.

The following subsections describe the base behaviors offered in DI-Guy AI.

3.3.1 Travel Behavior

Agents with this behavior move back and forth along a set of waypoints in the environment. While the techniques for manipulating the path shapes and component waypoints are the same as traditional DI-Guy paths, the way the path shape is used is quite different than how a path shape is used by a traditional people-blitzed DI-Guy character. Agents don't stay exactly on the path, but instead just use it as a

suggestion of the route they should follow, deviating from the path as necessary to avoid obstacles and other agents. Multiple Agents can share a single path.

This behavior is good for creating pedestrians walking along a sidewalk or guards on patrol.

Effect of Behavior Region and Behavior Path Shape

Since the Agents are moving back and forth along a path shape, the behavior path shape is very important. The behavior will not work without it.

The behavior region's boundaries have no effect unless they are solid.

Effect of Focus Character and Focus Group

Traveling characters will look at focus characters and groups.

3.3.2 Path Follow Behavior

Agents with this behavior use a path (note, not a path shape) as their route. Note that the path they follow is the specific character's individual path (agents cannot share paths currently). Because they are following the path, the agent will also perform the actions that are specified on the path. Agents using path follow that are then given a new behavior, such as flee, are able to rejoin the path and continue.

This behavior is good for controlling agents where you have a set of actions you want the character to perform while following a path – and also good if you expect that agent to get interrupted by an event. It works best when invoked by a script using `agent_path_follow("pathname")`.

Effect of Behavior Region and Behavior Path Shape

none

Effect of Focus Character and Focus Group

none

3.3.3 Wander Behavior

Agents with this behavior select random points in their behavior region (or waypoints on their optional path shape - see below), move to those points, and wait there for a time before selecting a new wander target.

This behavior is good for creating a group of people milling about in an area. Depending on the Agents' current variants, the result can look like calm people in a market area or a crowd of protesters.

Effect of Behavior Region and Behavior Path Shape

If a behavior region is specified, the wander targets will be selected from that region. If a behavior region is not set, but a behavior path is, the wander targets will be random waypoints on the behavior path.

Effect of Focus Character and Focus Group

When Agents have reached their wander targets, they will turn toward the current focus character during the stationary portion of their wander.

3.3.4 Mingle Behavior

This behavior is similar to Wander, but instead of selecting random points in the behavior region, Agents with this behavior find other like-minded minglers and select the same wander target and stay there for a random amount of time. After this random interval the Agent will try to find a new mingle target to avoid clumping.

Effect of Behavior Region and Behavior Path Shape

Same as for the Wander behavior when the Agent decides to find a new wander target.

Effect of Focus Character and Focus Group

When Agents have reached their mingle targets, they will turn toward the current focus character.

3.3.5 Flee Behavior

Agents with this behavior will try to move away at their fastest speed away from a specified position in the scene or specified character or group. Once they are far enough away from the flee point, they stop.

This behavior is good for having Agents react to and run away from a danger such as a nearby explosion or a character firing a weapon. It is best to use this behavior by invoking it via a script.

Effect of Behavior Region and Behavior Path Shape

The behavior region's boundaries have no effect unless they are solid. The Agent just tries to move directly away from the flee point.

Effect of Focus Character and Focus Group

Once the Agent has moved far enough away from the flee point so that it can stop, it will turn to face the current focus character.



3.3.6 Pursue Behavior

Agents with this behavior attempt to reach and stay within a specified distance of another Agent.

This behavior is good for having a group of people follow a leader controlled by some external source, such as an I-Guy being driven by a scenario participant.

Effect of Behavior Region and Behavior Path Shape

The behavior region's boundaries have no effect unless they are solid. The Agent just tries to move toward the pursue target.

Effect of Focus Character and Focus Group

The current focus character is the character the Agent will follow. Alternatively, if a focus group is specified, the agent will follow the nearest character in the specified group, periodically updating their focus target.

3.3.7 Attack Behavior

This behavior is similar to Pursue, but once characters move close enough to their target, they will attempt to attack that target, typically by firing their weapon at it. Note that you should set the variant to "aim", or a similar solution, so that the weapon will fire properly.

Effect of Behavior Region and Behavior Path Shape

Same as for the Pursue behavior, the behavior region's boundaries have no effect unless they are solid. The Agent just tries to move toward the attack target.

Effect of Focus Character and Focus Group

Same as for the Pursue behavior; the current focus character is the character that will be attacked. If the current focus character is killed, a new current focus character will be selected, if possible. Alternatively if a focus group is specified the agent will pursue and attack the nearest character in the specified group.

3.3.8 Idle Behavior

This behavior allows you to build your smarts on top of it. Idling agents still respond to variant and focus changes, but unlike other AI base behaviors where the Agent is periodically supplied a new desired location, that now becomes the responsibility of the author. Idling agents still act as dynamic obstacles to other agents.

When using a DI-Guy Character Path (path mode)

Not recommended. Use "None" behavior.

Not Following a DI-Guy Character Path (free mode)

Use functions like `diguyCharacter::set_desired_position()` or `diguyCharacter::move_to_point()` to command new locations for the Agents.

Effect of Behavior Region and Behavior Path Shape

The behavior region's boundaries have no effect .

Effect of Focus Character and Focus Group

No effect.

3.3.9 None Behavior

None is actually a very important behavior. It is typically used by an Agent to turn off his AI and instead use other means of controlling behavior, including DI-Guy paths or direct calls to his API. Note that if you want other Agents to avoid a DI-Guy character who is on a path, you at a minimum need to make him an Agent – and often you do this and specify a "None" behavior.

When using a DI-Guy Character Path (path mode)

Agents will be controlled by their character path, which determines their positions and actions as is typical for traditional DI-Guy characters. The Agents do, however, affect other Agents in its crowd, potentially causing them to flee, change behavior, get pushed out of the way, etc.

Not Following a DI-Guy Character Path (free mode)

None behavior will respond to commands like `force_action()`, but commands that internally use guides like `set_desired_position()` will not respond.

Effect of Behavior Region and Behavior Path Shape

The behavior region's boundaries have no effect.

Effect of Focus Character and Focus Group

When agents have reached their desired position they will turn toward the current focus character.

3.4 Regions and Sub-regions

Regions are irregularly shaped, 2-1/2 D, non-manifold areas in the scene associated with Crowds. They are most commonly created as a result of a Crowd Blitz or by using the Region Painter in the 3D View. Visually, they are represented as colored meshes covering the ground.

A region may contain a number of sub-regions. Each sub-region is represented by a different color, and most often the sub-regions are referred to by their color. Sub-regions are not mutually exclusive; a point in the scene may be in the base, populate, and green regions, for example.

The sub-regions are:

- base (visually represented as light blue)
- populate (visually represented as purple)
- red
- green
- blue
- yellow
- orange
- white
- road
- sidewalk
- crosswalk

The “non-color” regions have special meanings. The base sub-region underlies all other sub-regions – it is the sum of all the sub-regions. If a sub-region is expanded into an area not formerly part of the region, the base sub-region will be expanded, too.

When a Crowd is blitzed into the scene, it randomly places the new Agents in the populate sub-region.

The remaining “color” regions are free to be used in any way a scenario designer wants.

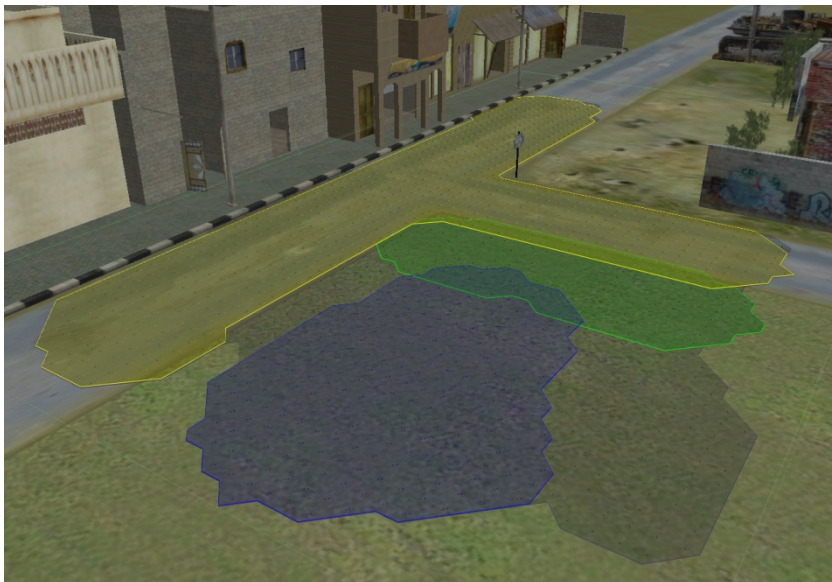


Figure -1 - A Region and its subregions

In the image above, a single region and its four sub-regions are shown: 1) the base, which represents the sum of all sub-regions (note that erased sub-regions will often be remembered by the base); 2) yellow,

which has been painted onto the streets and may serve as areas safe for vehicles but dangerous for humans; 3) green, an area where, for example, people waiting for a bus might stand; and 4) blue, an area safely away from the street.

Regions play a number of important roles in DI-Guy AI:

- The region painted in during a Crowd Blitz becomes the populate sub-region where Agents will be randomly placed.
- A number of Agent behaviors are influenced by the Agent's current region and sub-region. For example, Agents in the wander behavior select their wander targets from their current region and sub-region. The Agents can be easily moved to a new area by changing the current sub-region so that new wander targets will be in a new area.
- By default, regions do not limit an Agents' movements. However, the "walls" of regions can be made solid, to keep Agents inside the regions' limits.

3.5 Action Selection

The actions Agents will perform are determined in a number of ways. Some factors include how far away the Agent is from its desired position and proximity to other Agents, particularly those that have strong repulsion factors.

3.5.1 Speed Types

Because the set of actions varies for different DI-Guy character types, actions are broken down into higher-level "Speed Types". Rather than specifying the exact action an Agent should perform when it is in a particular Speed Zone, Speed Zones instead have an associated Speed Type. The Speed Types are:

- Still
- Slow
- Medium
- Fast
- Fastest

The exact action a character will perform boils down to a query to that character type's motion engine, with parameters to the call being the Speed Type, posture, and/or variant. Postures and variants are discussed in detail below.

Speed Types are used for action selection in both Speed Zones and Repulsion Zones (a dynamic object avoidance method).

3.5.2 Speed Zones

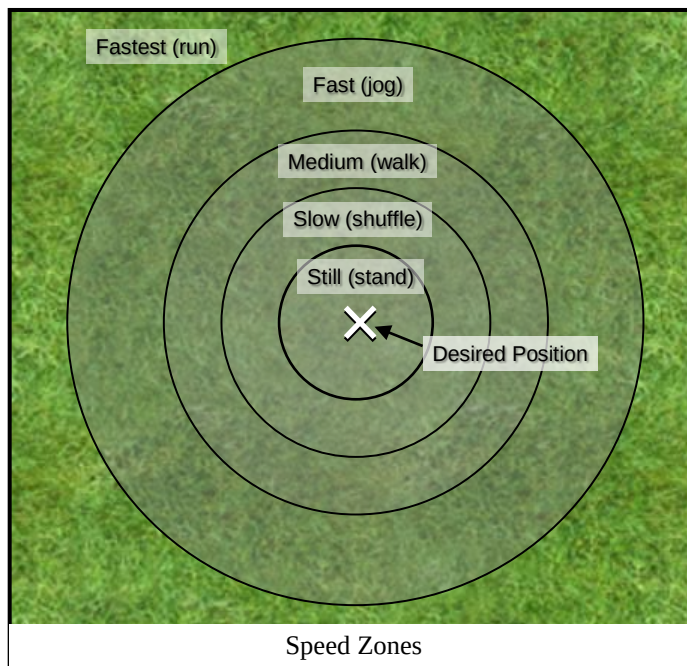
For most behaviors, the Agent has a desired position it is trying to reach, whether it is a new wander location, a character it is pursuing, or one explicitly set via the DI-Guy API. The action the Agent selects

usually depends on how far away the Agent is from the desired position; when it is far away, a faster speed type will be selected.

DI-Guy AI Agents use Speed Zones to specify the distances at which various speed types are invoked. There is therefore a one-to-one correspondence between Speed Zones and Speed Types.

The figure below shows the Speed Zones an Agent may be in, the Speed Type associated with the zone, and the resultant action for that particular character type that is used in that Zone.

(Note that Repulsion Zones of other Agents may also affect the Agent's final Speed Type. Repulsion Zones are discussed in another section.)



Speed Zone Parameters

Each Speed Zone has the following parameters:

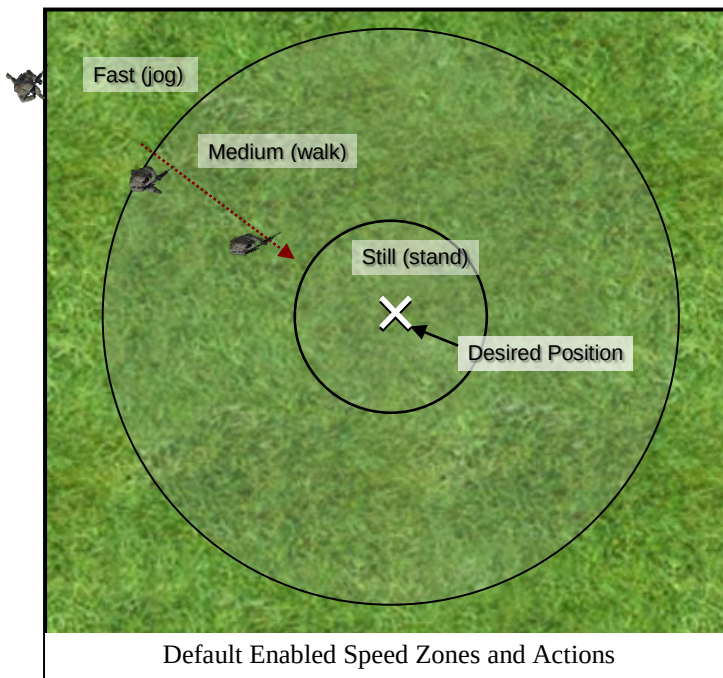
- **Enabled:** Whether the Speed Zone is enabled. Agents will ignore disabled zones, using instead the next faster zone. The Still Speed Zone is always enabled.
- **Start At Radius:** Radius of Speed Zone circle, centered on the Agent's desired position. The speed zone starts at this radius, extending outward to the next enabled zone's Start At Radius. The Start At Radius of the Still Speed Zone is always zero.

- **Exact Action:** An optional exact action that the Agent will use in the zone. This action overrides the action typically calculated via the desired Speed Type, posture, and variant.

Disabled Speed Zones

The Slow and Fastest zones are usually disabled, meaning that Agents tend to use only their jog, walk, and stand actions. The Slow zone typically has a 2 meter radius; the Fast zone 10 meters.

In practice this means that Agents further than 10 meters from their desired position will move at their fast speed, which for the normal variant is commonly a jog. Agents between 2 and 10 meters from their desired position will move at their medium speed, which for the normal variant is commonly a walk. Agents within 2 meters will be still, which for the normal variant is commonly a stand motion.



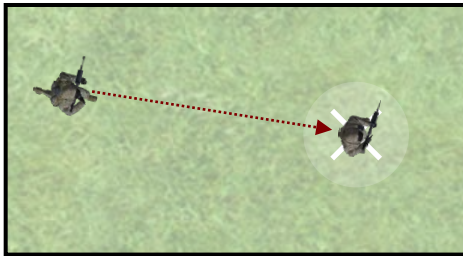
The Still Speed Zone

The Still Speed Zone radius is usually a meter or more. Note that this means that Agents will rarely reach their exact desired position. To understand why, try the following thought experiment:

Imagine that you are standing by yourself in an empty field. On the ground in the center of the field, about ten meters away, is a coin. You are listening to and following directions from a coach on the side of the field.

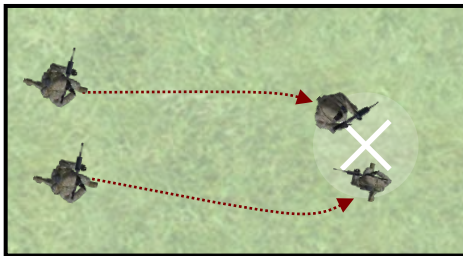


The coach tells you, “Go to the coin.” Chances are, you will go and stand directly on top of the coin.



Now imagine that you’re again ten meters from the coin, and there is another person with you. The coach tells both of you, “Go to the coin.” There are a couple of things that can happen this time.

1. Both of you move to a small distance away from the coin and stop.
2. The person who gets there first stands on top of the coin. The second person stands some small distance away.
3. Least likely, unless both of you are remarkably good friends, you both stand on the coin, one somehow on top of the other.



Imagine this scenario again, but this time there are dozens of people trying to get to the coin. This time almost no-one will be able to stand on the coin, and some will be unable to get close to the coin.

In general, the more Agents there are that try to reach the same desired position, the bigger the Still Zone needs to be.

Speed Zone – the Dangers of equal Still Zone radii and the Radius Variation Parameter

Very often, all Agents in a Crowd have the same Agent parameters, as it's much easier to edit the parameters of Crowd Agents as a whole rather than on an Agent-by-Agent basis. However, if all Agents in a Crowd have the same Speed Zone radii, some unnatural-looking behavior can result.

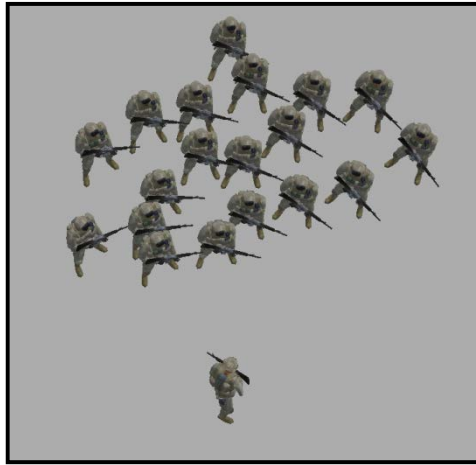
Consider the example of dozens of Agents attempting to reach a single desired position, in this case another Agent they are pursuing. If all of the Agents have the same Still Speed Zone radius, then all of them will stop at the same distance from the pursuit target, resulting in something like this:



All of the Agents are pretty close to six meters from the pursuit target, with six meters being the Still Zone radius. In many cases this doesn't look realistic.

What would look better is if each Agent had a slightly different radius for each Zone. That can be done by using the *Radius Variation* parameter. If the variation is non-zero, a random adjustment between zero and the variation value is added to each Zone radius.

For example, if the Still Zone radius is two meters, and the radius variation is four meters, then Agents in the pursuing Crowd will effectively have a Still Zone radius between two and six. That's two, plus a random number between zero and four. The result looks like this:



This looks a bit more natural.

3.5.3 Desired Posture and Desired Variant

Each DI-Guy action has a *posture*. The postures that DI-Guy AI Agents make use of are:

- upright (the most common) (example actions: “stand”, “walk”, “run”)
- crouched (example actions: “walk_lo”, “kneel_aim”)
- prone (example actions: – “crawl”, “prone_aim”)

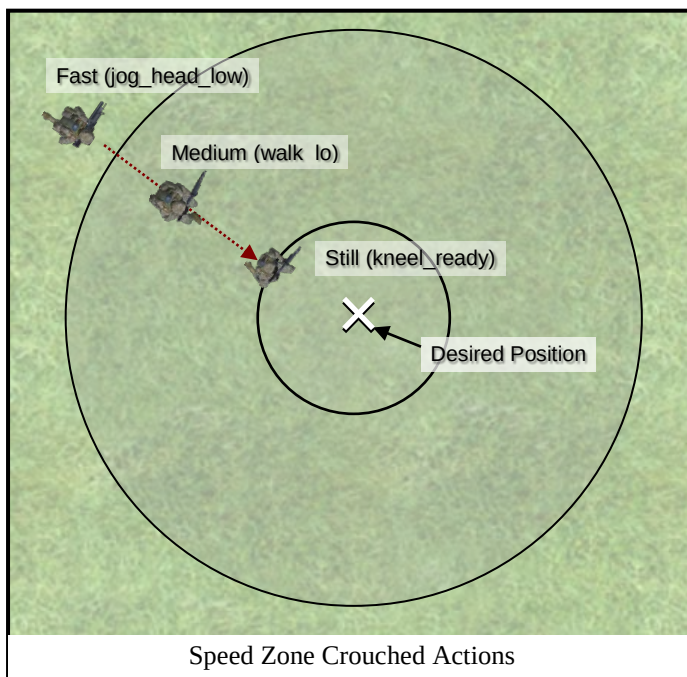
In addition to posture, each action has an associated *variant*. Variants provide another way of differentiating actions that otherwise are pretty much the same. For example, the actions stand and talk are very similar: both are upright, both face forward, both are still. In this case, the variant can be used to differentiate between the two. The stand action has the variant “normal”; the talk action has the variant “socialize”.

The variants that DI-Guy AI Agents make use of are:

- normal (the most common)
- aim
- ambient (the 2nd most common)
- dead
- ready
- injured
- sick
- exercise
- angry
- curious
- socialize

An example of an upright posture, normal variation action would be a “stand”. An example of an upright posture, *angry* variant action would be an “angry_chant” or “rock_throw” action.

The desired posture and variant of an Agent can have a strong effect on which actions are selected based on the Agent's current Speed Zone. For example, if the desired posture is set to “crouched”, the actions selected for each Speed Zone would be the following:



The desired variant can have a similar effect. If the desired variant were set, say, to “socialize”, then when the Agent arrives at the Still Speed Zone it may begin the action “talk1” instead of the default stand.

3.5.4 Automatic Variant Selection

Introduced in DI-Guy 12.5 we've added a system for automatically picking appropriate variations for agents to limit the likelihood that Agents will wind up standing around like statues. This is a simple low level C++ system that doesn't require the use of Lua.

The rules for the picking the variant are as follows:

Behavior	Variant
Attack	Has Target: Within attack distance plus a bit: <i>Aim</i> Outside that distance: <i>Normal</i> No Target: <i>Ambient</i>
Idle	<i>Ambient</i>
Mingle	With Focus Character: <i>Socialize</i> Alone: <i>Ambient</i>
Wander	<i>Ambient</i>
Flee	With Focus Character: <i>Curious</i> Without: <i>Ambient</i>
Pursue	In a formation (ie using a pursue offset): <i>Normal</i> With Focus Character: <i>Curious</i>

This system can be disabled by calling `diguyCharacter::agent_set_auto_variant_selection()`

3.6 Obstacles Overview

Sometimes there are obstacles in an Agent's way. When that happens, the Agent has some decisions to make, depending on what the obstacles are. If the obstacle is a simple obstruction such as a light pole, the Agent can simply move around it. If, however, it is a nearby gunman who is firing his weapon, the Agent will need to take a more drastic evasion approach.

DI-Guy AI breaks obstacles down into two kinds. The first are those that never move, such as buildings, walls, and fixed props in the scene. These are called static objects. The second are moving obstacles, such as humans and vehicles. These are called dynamic objects.

3.6.1 Static Object Sensing and Avoidance

Static objects are objects that do not move. There are currently three modes Agents can be in with respect to static object detection and avoidance:

- Disabled
- Feelers
- Path Planning

While “Disabled” is not really a method but rather the lack of one, it is still an important consideration.

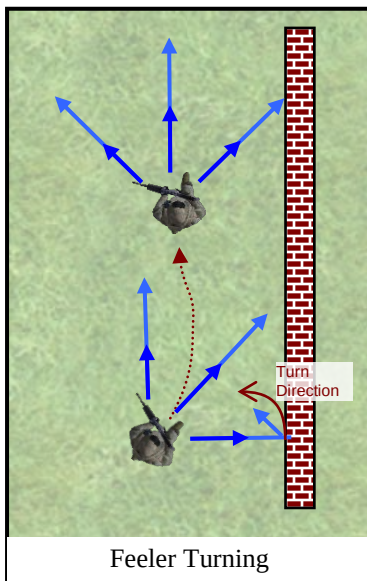
3.6.2 Feeler Static Avoidance Method

One static object avoidance method DI-Guy AI Agents use is based on “feelers”. Feelers are rays that project from the Agent’s body, looking for intersections with static objects in the scene. An Agent can have up to eight feelers. One common configuration is to use two: one projecting to the front and left, the other projecting to the front and right. The best choice needs to balance performance (feelers cost CPU) and robust collision detection.

If a feeler touches something, the Agent will respond. If the touched object is still some distance away (ie – between the turning and collision distances), the Agent will turn to move along the object.

If the object is closer, that is, less than the collision distance, the Agent determines he is colliding with the object and will forcibly away from it.

The following figure shows an Agent approaching a wall. The front right feeler touches the wall. Since the wall is within the turn distance but outside the collision distance, the feeler turns the Agent to move along the wall.

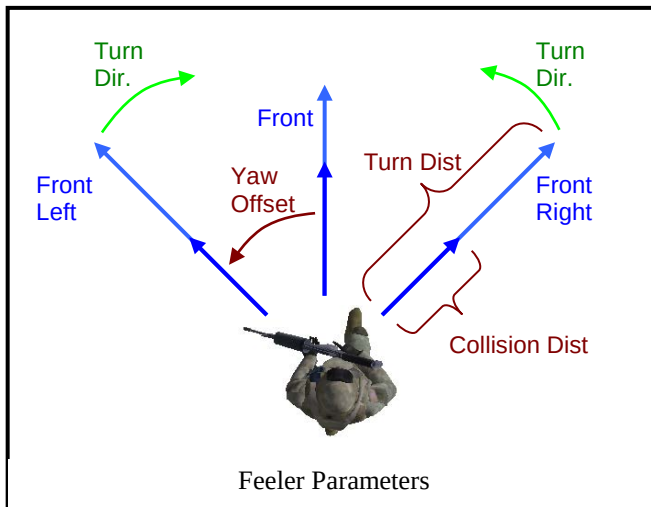


Feeler Parameters

Feelers have a number of tunable parameters:

- **Enabled:** Whether the feeler is enabled. Disabled feelers will have no effect on the Agent. By default only the front left and front right feelers are enabled.

- **Yaw (rz) Offset:** The yaw (rotation about Z axis, or rz) offset of the feeler from forward. The default eight feelers are evenly spaced at 45 degree intervals around the Agent.
- **Height:** The height of the feeler from the ground. By default the height is 0.7 meters.
- **Collision Distance:** The distance at which the feeler returns a collision result, and moves the Agent away from the object. By default this is 0.25 meters.
- **Turn Distance:** The distance at which the feeler returns a turn result, and turns the Agent to move along the object. By default this is 0.5 meters.
- **Turn Direction:** Which direction the feeler will turn the Agent if contact occurs inside the turn distance. By default the front left feeler turns the Agent right, the front right feeler turns the Agent left, and the front feeler turns the Agent left or right depending on the angle of contact.



3.6.3 Disabled Static Avoidance Method

Static object avoidance can be disabled. In this case static objects are not detected or avoided, which in effect this means that characters can walk through walls and solid objects.

If the Agent is not near any static objects this isn't a problem, and disabling static object avoidance can reduce the performance overhead.

3.6.4 Dynamic Object Sensing and Avoidance

Dynamic objects are objects that move around the scene. Almost always these are other Agents, but can also be incoming DIS/HLA entities or avatars. There are currently two methods Agents can use to detect and avoid them:

- Repulsion Zones
- Disabled

While "Disabled" is not really a method but rather a lack of one, it is still an important consideration.

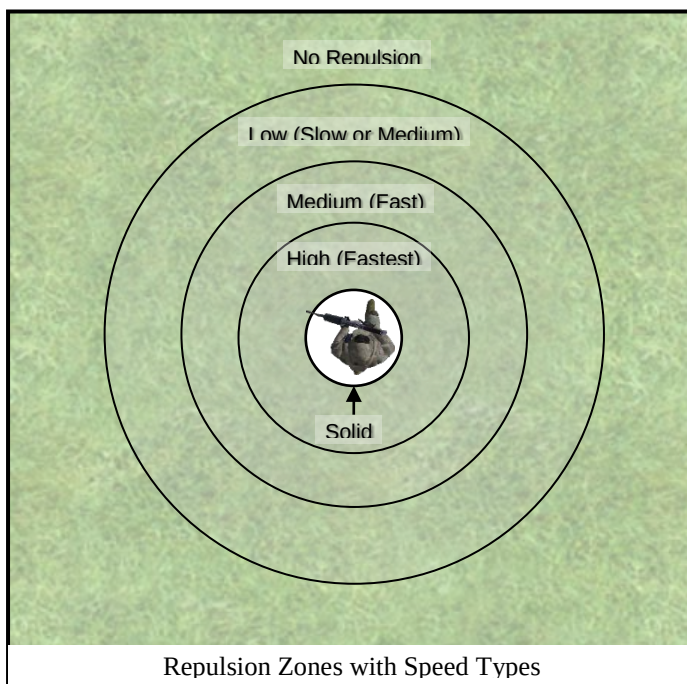
3.6.5 Repulsion Zones Dynamic Avoidance Method

In this method Agents are surrounded by zones of repulsion. Other Agents that enter these zones have their position, orientation, and/or action changed to try to get out of them.

Note that these zones are different than the Speed Zones discussed in the Action Selection section. An Agent's Repulsion Zones are centered on the Agent, and affect how *other* Agents act. An Agent's Speed Zones are centered on the Agent's desired position, and affect only how *this* Agent acts.

Each Repulsion Zone has a number of tunable parameters:

- **Enabled:** Whether the zone is enabled. Disabled zones will have no effect on other Agents. By default the Medium and High Repulsion Zones are disabled.
- **Repulsion Radius:** How far from the Agent's position the zone extends.
- **Repulsion Factor:** How strong a "force" should be applied to Agents that enter the zone.



Agents in another Agent's Repulsion Zone will be affected in a couple of ways.

Orientation

A "force" will be applied to the Agent that attempts to turn the Agent directly away from the zone center (which is the position of the Agent owning the Repulsion Zone). At the same time, another "force" may be being applied to turn the Agent toward its desired position. In addition, the Agent

may be in multiple Repulsion Zones if it is in close proximity to multiple Agents. These will all exert an orientation “force”.

All of these “forces” are combined together to determine a final change to the Agent’s orientation. The Agent does have a maximum orientation change rate, so regardless of the forces being applied it will never exceed the maximum.

Position

An Agent entering the Solid Zone of another Agent will be moved out of it.

Action Selection

Each Repulsion Zone has an associated Speed Type.

Agents in a Low Repulsion Zone will try to avoid at a low speed, either Slow or Medium.

Agents in a Medium Repulsion Zone will try to avoid at their Fast speed, which is typically a jog.

Agents in a High Repulsion Zone will try to avoid at their Fastest speed, which is typically a run.

Speed Zone and Repulsion Zone Interactions

Note that there is a lot of overlap in selecting the proper speed between the Speed Zones described previously, and these Repulsion Zones. Both are used to determine the Speed Type the Agent will use. In general, the fastest Speed Type is used.

If, for example, an Agent is in its Fastest Speed Zone and in the Low Repulsion Zone of another Agent, the final Speed Type will be Fastest. Similarly, if an Agent is in its Slow Speed Zone but is in the High Repulsion Zone of another Agent, the final Speed Type will be Fast.

3.6.6 Repulsion Zones and Vehicles

Human-to-vehicle collision detection and avoidance is different than human-to-human in DI-Guy AI.

Difference in Mass

Vehicles are heavier than humans. While it is possible for a human to push another human out of the way, humans cannot push a vehicle out of the way. Because of this, forced position change due to Solid Zone penetration is one-sided, with only the human being moved.

Difference in Speed

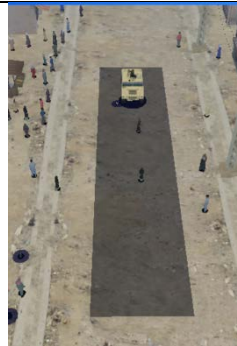
Vehicles can move extremely fast compared to humans, therefore vehicle-to-human collisions are much more serious than human-to-human ones. Because of this, vehicle Repulsion Zones can stretch much further in front of them than a typical human’s does.

Bounding Shape

Vehicles do better with box-like bounding shapes and Repulsion Zones rather than the circular ones of humans. Because of the needed larger Repulsion Zones mentioned above, if the zones were circular they would extend much too far to the sides and in back of the vehicle. Instead, vehicle Repulsion Zones extend fairly far in front of the vehicle and comparatively little to the sides.



A slow moving vehicle and it's influence



A fast moving vehicle and it's influence

3.6.6.1 Disabled Dynamic Avoidance Method

Dynamic object avoidance can be disabled completely, in effect enabling crowd members to walk through each other.

If a Crowd is far away from the viewpoint it may not be obvious that this is happening. Disabling a Crowd's dynamic object avoidance can enhance performance.

3.7 Detecting and Getting Out Of Traps

Agents not using A-star path planning will sometimes find themselves in a dead end, or otherwise unable to make progress toward where they want to go. DI-Guy AI refers to such an Agent as being “trapped”. There are a number of methods that can be applied in order to attempt to “untrap” the Agent.

In some cases DI-Guy AI Agents do not always have global knowledge of their environment. That is, the entire world is not mapped out offline with all possible paths computed from point to point via A-star. The only information available to Agents is then:

- Their own position, orientation, and desired position.
- Polygons of static objects that are within their feeler ranges.
- Other Agents in their Crowd or Companion Crowds.

Because of this it is not uncommon for Agents to end up stuck in a local minimum; that is, in a boxed-in area in which the standard approach of using feelers to move around obstacles is not sufficient.

3.7.1 Trap Detection

Agents currently detect two kinds of traps: being “wedged”, and a “no progress” trap.

Wedged Trap

This trap occurs when the Agent's feelers are detecting continuous collisions for an amount of time greater than the wedged trap time threshold. Most often this occurs when the Agent has

navigated into a tight corner, or is being pushed into an area by the repulsion effects of other Agents.

No Progress Trap

This trap occurs when the Agent is apparently freely able to move around the scene, but something has been keeping it from making real progress toward its desired position for some time. This is commonly when the Agent gets trapped in a local minima area.

Agents check for No Progress traps at two different time thresholds. Different untrap methods (see next section) can then be applied, first a benign untrap method and then if that doesn't work a more drastic untrap method.

3.7.2 Untrap Methods

Random Turn

The Agent will temporarily stop moving, turn a random amount, and then resume movement. This minor course change can sometimes move a character around a small obstacle.

Ghost

The Agent's feelers will stop having an effect for a short time. This allows the Agent to walk through a wall toward its desired position. (If the Agent is directly in front of the viewpoint this can be a little ugly, but may be better than the Agent hopelessly walking into a corner for many seconds.)

Nav Path

The Agent will try to create a navigation path ("nav path") in its current region to its desired position, along which the Agent can move. This is in effect temporarily puts the Agent in the travel behavior.

This method will not work unless the Agent, the desired position, and a navigable path lie entirely in the Agent's base region.

Teleport

This is an extreme method, typically used as a last resort. The Agent sets its current position to be the desired position, resulting in a visual discontinuity, in effect magically disappearing from its current position and reappearing at its desired position instantly.

None

The Agent makes no attempt to become untrapped. This can result in the Agent stubbornly walking in a corner until its desired position changes.

Stop Moving

Somewhat similar to the None method above, but the current behavior of the Agent is set to "none", and the Agent character's desired position is set to its current position. Unless something else is sending desired position updates to the Agent, this should cause the Agent to stop at its current position and wait for further instructions.

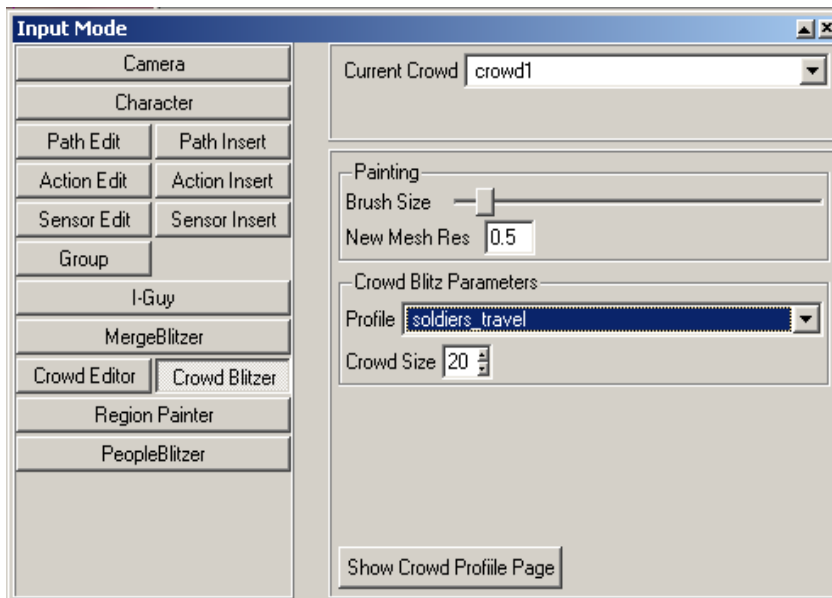
DI-Guy AI GUI Overview

This Section describes in detail the pages and tabs of DI-Guy Scenario with DI-Guy AI.

As you go through the descriptions below, remember to go back to Chapter 3 DI-Guy AI Concepts, when wanting to better understand the various building blocks of DI-Guy AI.

3.8 Input Mode Palette

The Input Mode palette controls what the mouse does in the 3D window when working with DI-Guy Scenario. DI-Guy AI adds three new modes to the Input Mode palette: Crowd Editor, Crowd Blitzer, and Region Painter.



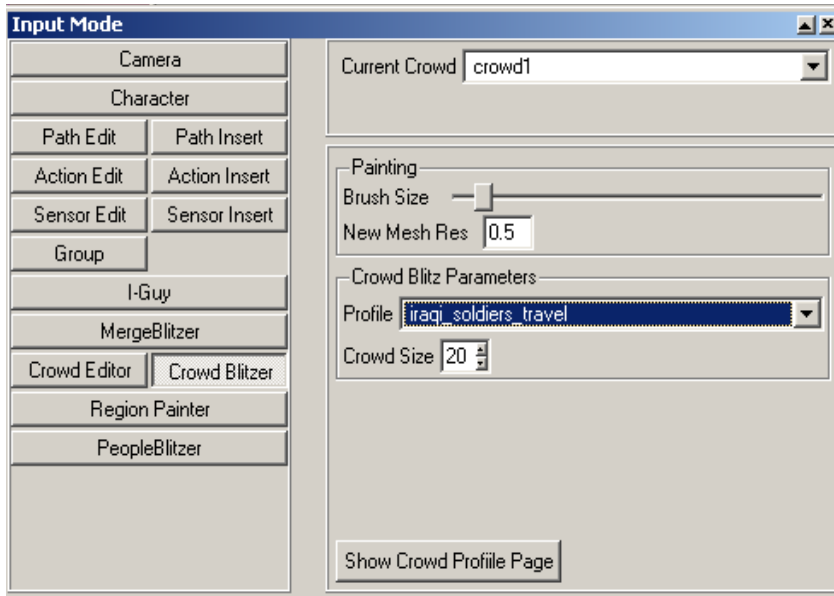
the Input Mode Palette

3.8.1 Crowd Blitzer



A large crowd of pedestrians is painted into the 3D Display using Crowd Blitzer.

Crowd Blitzer is a powerful input mode for quickly creating a large Crowd of DI-Guy characters. Simply click on Crowd Blitzer on the Input Mode palette, choose your settings, and then brush with the 3D brush in the 3D display; a new crowd of DI-Guy characters, based on the chosen Crowd Profile, is created. Hit “play” on the Playback Controller palette to immediately see the characters perform their tasks.



The Input Mode palette with Crowd Blitzer selected.

Following are brief descriptions of the input parameters to Crowd Blitzer:

- **Brush Size** – modify the size of the 3D brush when blitzing in the population region. Use a small brush for sidewalks and other narrow areas. Use a big brush to fill in streets and fields with Agents. Note that the spin wheel on the mouse will also let you alter the brush size.
- **Mesh Resolution** – Regions in DI-Guy AI are defined by a regular mesh of points. If you look closely, you can see these points in the mesh graphic. The finer the mesh, the more likely characters will find their way into nooks and crannies in the database, but too fine a mesh and more memory will be consumed than necessary.
- **Profile** – choose a Crowd Profile, which describes all the basic features of the new Crowd to be blitzed including the base behaviors, whether the character use Minds, and the appearances of the characters. Note that Crowd Profiles are defined in the DI-Guy AI Objects|Crowd Profile page, where you can also define your own.
- **Crowd Size** – this will default to the value specified in the Crowd Profile, but you can override it to create the exact size crowd you want.
- **Show Crowd Profile Page** – A shortcut button for displaying the Crowd Profile tab of the AI Elements page.

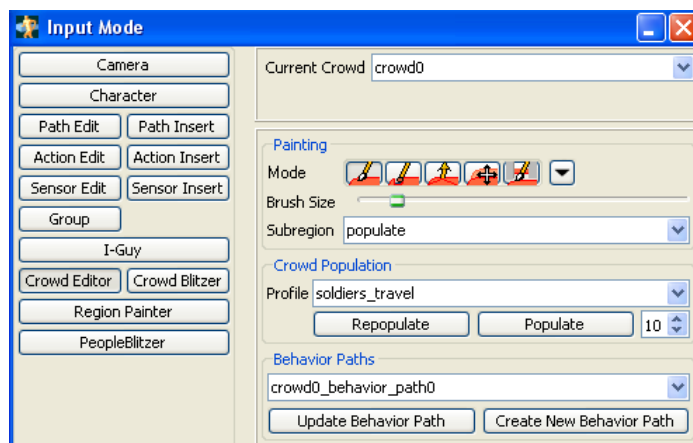
DI-Guy AI Tip: For some behaviors, such as travel, the underlying Behavior Path Shape is important. When you paint in your region, try to do it with a single brush stroke, as the click down and click up mouse button points will become the start and end points of the Behavior Path Shape.

3.8.2 Crowd Editor



A large crowd of pedestrians is edited using the Crowd Editor.

The Crowd Editor input mode is used when you want to modify an existing Crowd. The parameters and methods for doing so are similar to Crowd Blitz. Note that the DI-Guy AI Objects|Crowd page is also a place where you can change crowd parameters.



The Crowd Editor is selected in the Input Mode palette.

Here is a description of the parameters specific to Crowd Editor:

- Current Crowd – use this pulldown to select between the different Crowds you have created.

Crowd Editor Painting Parameters

- Painting Mode: This lets you select between 5 different edit modes when working with your Crowd.
 - Paint: Add to your existing region by painting.
 - Eraser: Subtract from your existing region by painting. Note that holding down the Ctrl key while in Paint Mode will also activate the Eraser.
 - Unground Clamp: Erase the ground clamping in the Region you paint. This is useful when working with multi-story buildings and the Region has clamped to the wrong floor.
 - Move Mesh: Grab your Region with the mouse and move it around.
 - Paint Reachable region – Paint a “Reachable” subregion. By default, the paint brush dynamically analyzes the area around where you paint. Occasionally, while analyzing around certain areas like doors, the paint brush can be wrong. The reachable brush allows you to force an area to be reachable or non-reachable disregarding the dynamic analysis.
 - Painting Preferences Pulldown – This pulldown lets you alter some basic parameters when painting.

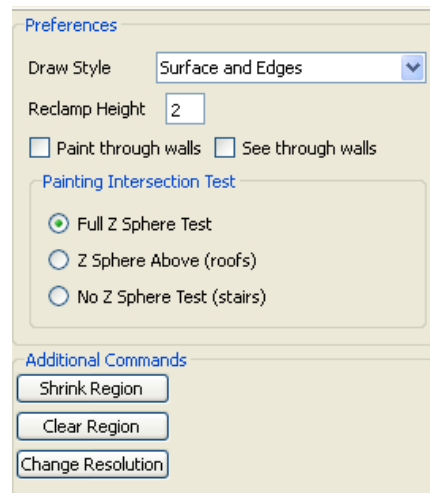


Figure -5 - The Paint Preferences Pulldown

- ❖ Draw Style. Select between “surface and points”, “surface”, and “points”. The surface is the colored interior region, while the edge is the perimeter drawn in a different color.
- ❖ Reclamp Height. This specifies the starting height from which ground clamping ray tracing is performed and defaults to 2 meters. It is useful, for example, if

you want to paint streets in an urban environment, you can set the Reclamp Height lower than the rooftops, only the streets will get painted.

- ❖ Paint through walls toggle. Normally, you want to leave “Paint through walls” disabled, so that, for example, you can paint a sidewalk right up to the sides of a building, but without painting on the roof of that building. Note that painting through walls does NOT mean that characters can walk through the walls.
 - ❖ See through walls. Graphics for regions will be displayed even when they would normally be hidden behind a wall (region drawing ignores z-buffer)
 - ❖ Shrink Region. This will shrink the extents box down to the extents of the current Region. This is useful for conserving memory.
 - ❖ Clear Region. This erases the Region.
 - ❖ Change Resolution. Modify the Region resolution. The region that you are painting is defined by a mesh of points. If you look closely, you can see these points in the mesh graphic. The finer the mesh, the more likely characters will find their way into nooks and crannies in the database, but you don’t want to use too fine a mesh and consume more memory than necessary.
- Brush Size – modify the size of the 3D brush for painting. Note that the mouse spin wheel will also let you alter the brush size. Use a small brush for sidewalks and other narrow areas, while using a big brush to fill streets and fields with DI-Guy characters.
 - Sub-region – Select the region associated with the crowd that you wish to edit. Normally this is defaulted to “populate”. See the Region Painter section below for a more thorough description of how multiple regions work with crowds.

Crowd Editor Crowd Population Parameters

- Profile – choose a Crowd Profile, which describes all the basic features of the Crowd from the characters that compose it to the behavior they will exhibit. Note that Crowd Profiles are defined in the DI-Guy AI Objects|Crowd Profile page, where you can also define your own. Altering this setting will change the composition of the entire crowd, and does not require a paint.
- Repopulate – This button will delete all the Agents from the Crowd and repopulate the Crowd with a whole new set of Agents. The number of Agents is specified by Crowd Size.
- Populate – This button will add more Agents to your existing crowd. For example, if you have a Crowd of 60 Agents, and you have the Crowd Size set to 10, Populate will add 10 more Agents, resulting in a Crowd of 70 Agents.
- Crowd Size – this value is used with Repopulate and Populate, and is seeded from the current crowd profile.

Crowd Editor Behavior Paths

- Current Behavior Path Pulldown: Select a behavior path. Some base behaviors, such as travel, associate a Behavior Path with the entire crowd. See section 3.3 for more information on Behavior Paths and how they interact with particular Behaviors.

- **Update Behavior Path:** Updates a behavior path based on changes made to the Region. DI-Guy AI automatically tries to plot the best Behavior Path Shape based on the Region's topology.
- **Create New Behavior Path:** Creates a new Behavior Path associated with the crowd.

Painting Hot Keys

We've added in some extra controls to help make crowd painting as easy as possible:

- o **SHIFT KEY** – Contiguous regions.
- o **CTRL KEY** – Activates the eraser.
- o **MOUSEWHEEL** – Alter paintbrush size.
- o **RIGHT CLICK** – Access a popup menu listing sub-regions.
- o **ALT** – disable painting and activate the camera (this is a general DI-Guy Scenario override)

3.8.3 Region Painter

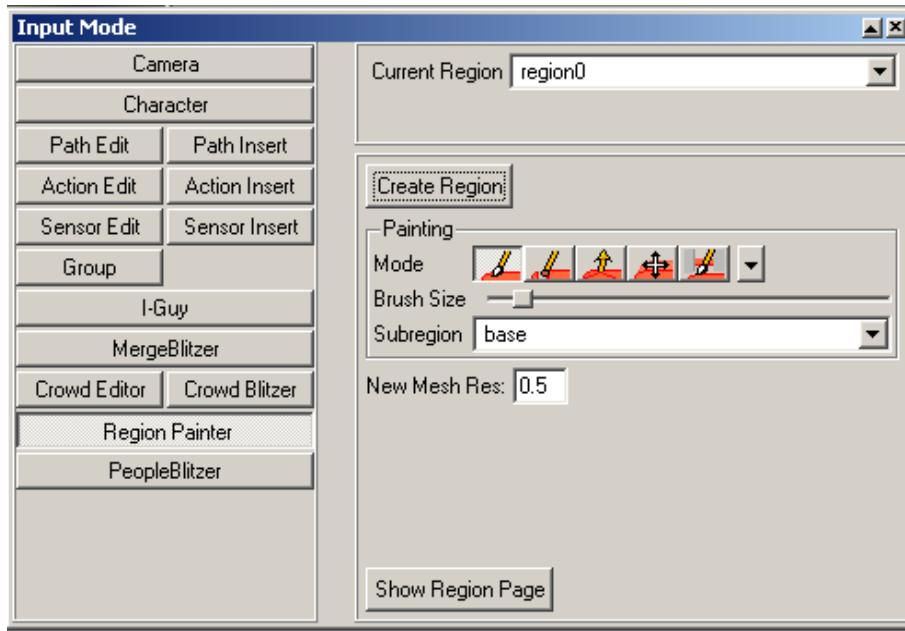


Figure -6 - Region Painter is selected in the Input Mode Palette.

A Region is a mesh of points in DI-Guy AI used to understand topology. Normally you use them by associating them with crowds, but they can be independent of crowds, and multiple crowds can share regions. When you Crowd Blitz in a crowd of DI-Guy characters, you implicitly create a region called the “populate” region. Crowds can have multiple regions associated with them (you can choose between “base”, “populate”, “red”, “green”, “blue”, “yellow”, “orange”, “white”, “road”, “sidewalk”, and “crosswalk”). Regions are used in conjunction with behaviors.

There is some redundancy between the region tools here and the ones in Crowd Editor for convenience. Feel free to use either tool.

See Section 3.4 for more information on Regions.

Following is a description of the parameters specific to Region Painter:

- Current Region Pulldown: Select a Region for painting.
- Create Region: Creates a new Region.

Region Painter Painting Parameters

- Painting Mode: This lets you select between 5 different edit modes when working with the current Region.
 - Paint: Add to your existing Region by painting.

- Eraser: Subtract from your existing Region by painting. Note that holding down the Control key in Paint Mode will also activate the Eraser.
- Unground Clamp: Erase the ground clamping in the Region you paint. This is useful when working with multi-story buildings, and the Region has clamped to the wrong floor.
- Move Mesh: Grab your Region and move it around.
- Paint Reachable region – Paint a reachable Region.
- Painting Preferences Pulldown – This pulldown lets you alter some basic parameters when painting. See section 3.8.2 for descriptions of the preferences.
- Brush Size – modify the size of the 3D brush to maximize ease-of-use of Region Painter. Use a small brush for sidewalks and other narrow areas, while using a big brush to create large regions.
- Sub-region – Assign the region type. Choices are (base, populate, red, green, blue, white, violet)
- New Mesh Resolution. Resolution when creating new meshes. The Region that you are painting is defined by a mesh of points. If you look closely, you can see these points in the mesh graphic. The finer the mesh, the more likely characters will find their way into nooks and crannies in the database, but you don't want to use too fine a mesh and consume more memory than necessary.
- Show Region Page – A shortcut button to bring up the DI-Guy AI Objects|Regions page.

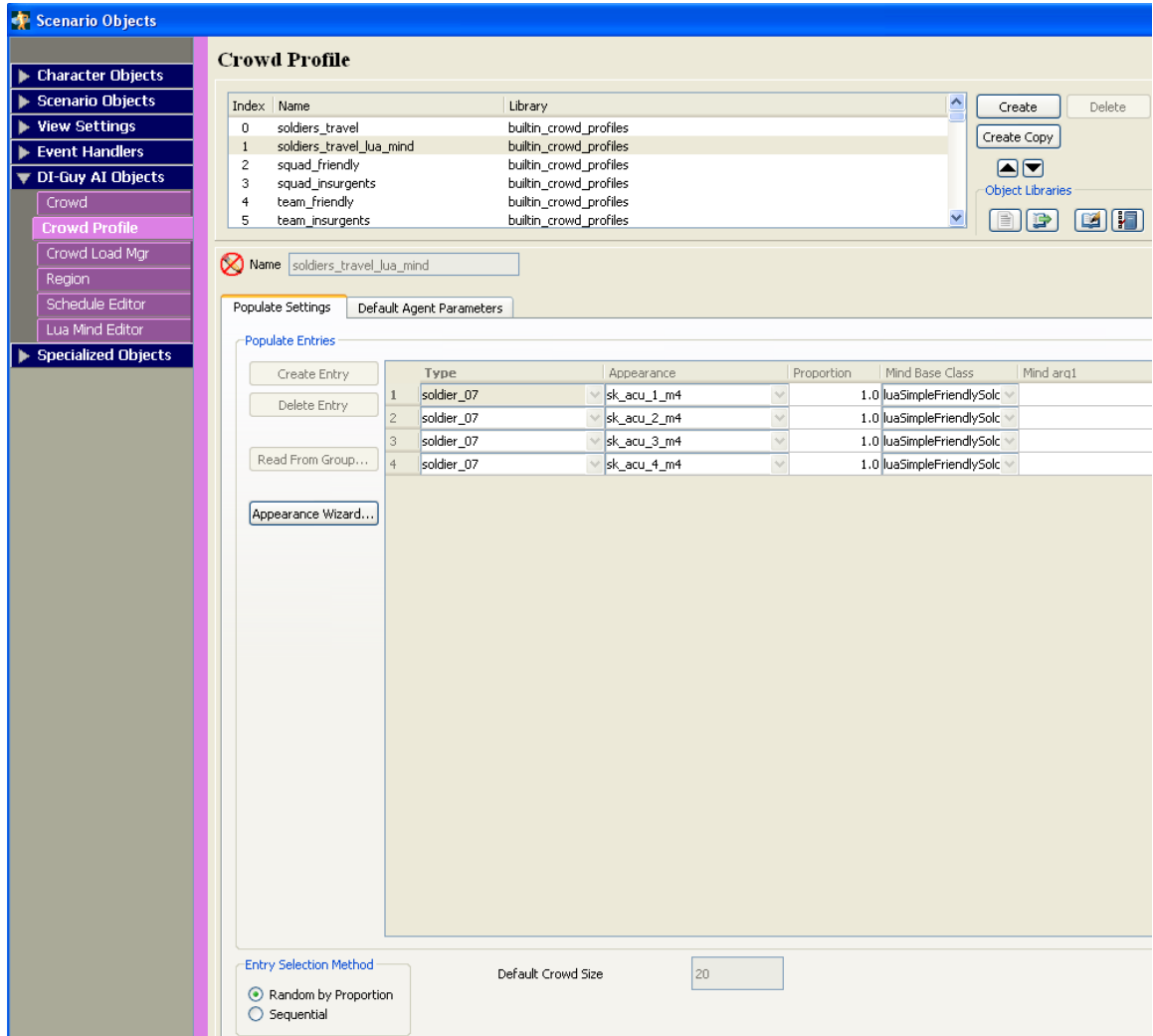
Painting Hot Keys

We've added in some extra controls to help make region painting as easy as possible:

- SHIFT KEY – Contiguous regions.
- CTRL KEY – Activates the eraser.
- MOUSEWHEEL – Alter paintbrush size.
- RIGHT CLICK – Access the subregions.
- ALT – disable painting and instead go to camera mode (standard DI-Guy Scenario override)

3.9 Crowd Profile

The DI-Guy AI Objects->Crowd Profile page is where you create basic Crowd descriptions that are used as input when Crowd Blitzing. DI-Guy supplies scenarios with some default Crowd Profiles, but you will soon want to create your own Crowd Profiles so that you can quickly populate your world with crowds of DI-Guy Agents that have the exact look and behavior you want.



The Crowd Profile Page

Here is a description of the parameters specific to the Crowd Profile Page:

- **Current Crowd Profile Select Box:** Select the current Crowd Profile. In the image above, “soldiers_travel_lua_mind” is chosen from a lengthy list of available crowd profiles.

- Create Crowd Profile: Push the “Create” button to create a new Crowd Profile.
- Delete Crowd Profile: Push the “Delete” button to delete the selected Crowd Profile.
- Create Copy Crowd Profile: Create a duplicate crowd profile to the selected Crowd Profile.
- Up and down keys: Move the current Crowd Profile up or down in the list.
- Object Library manipulators: The Object Library makes it easier to share profiles between scenario files. The four button to work with the Object Library are:
 - Display Object Text – show what the internal description of the current profile looks like. This is the code that is saved with the .dss file.
 - Copy Object Into Scenario – move an object from an external library such as “builtin_crowd_profiles” to a local copy.
 - Manage Object libraries – copy objects from local to libraries or between libraries.
 - Select Object Libraries to include in scenario – browse for new libraries to include. Useful when you want to share libraries across projects.
- Name Textbox: Rename the current Crowd Profile.

3.9.1 Populate Settings Tab

The Populate Settings Tab of the Crowd Profile Page is used mainly to designate what character types, appearances, and proportions from the DI-Guy content library will be used to compose a Crowd. Its main feature is the *Populate Entries Table*.

The Populate Entries Table

Each row corresponds to an entry, and each entry has 4 data fields.

The first data field is Character Type, which designates the DI-Guy character type of the entry. Remember that a DI-Guy character type designates 1) The set of actions a DI-Guy character can perform, and 2) the set of appearances available.

The second data column is Appearance. Here you can select the exact appearance of your Agent.

The third column is proportion. When a Crowd is blitzed in, each entry will be blitzed in according to its proportion. The proportion column is not constrained to sum up to a particular number and instead is used relative to the other proportions. It will simply sum up the total of all the proportions, and then populate based on the fractional amount each proportion is of the entire sum. (e.g – if you have 4 entries, 3 of which have proportion 1 and 1 that has proportion 2, then 40% of the population will be the proportion 2 entry, while the remaining 3 entries will be 20% each of the population)

The fourth data column is Mind Class. This indicates whether there is an associated DI-Guy AI Mind with the entry

The remaining columns are optional argument fields to the Mind class.

Each data field in the table is editable; simply use the pull-downs to select new character types and appearances, mind classes, and type in the proportions as necessary.

There are also 4 buttons associated with the Populate Entries Table:

- Create Entry – Append a new entry to the table.
- Delete Entry – Delete the current entry selected.
- Read from Group...- This button brings up a dialog box where you can surf through the DI-Guy Groups in your scenario, and create an entry corresponding to a member character in that group.
- Appearance Wizard – use this button to assist with determining character types and appearances for your entry. The wizard lets you enter descriptions of characters and returns back ranked suggestions for which character type and appearance to apply.

Entry Selection Method

These toggle buttons let you select how the Populate Entries Table will be used by the Crowd Blitzer.

- Random by Proportion – The members of your Crowd will use the proportions as a guideline, but will mix up the order of the entries.
- Sequential – The members of your Crowd will be created to match the entries in the Populate Entries Table in the same order.

Default Crowd Size

- The number of entities a crowd will have when using this Crowd Profile.

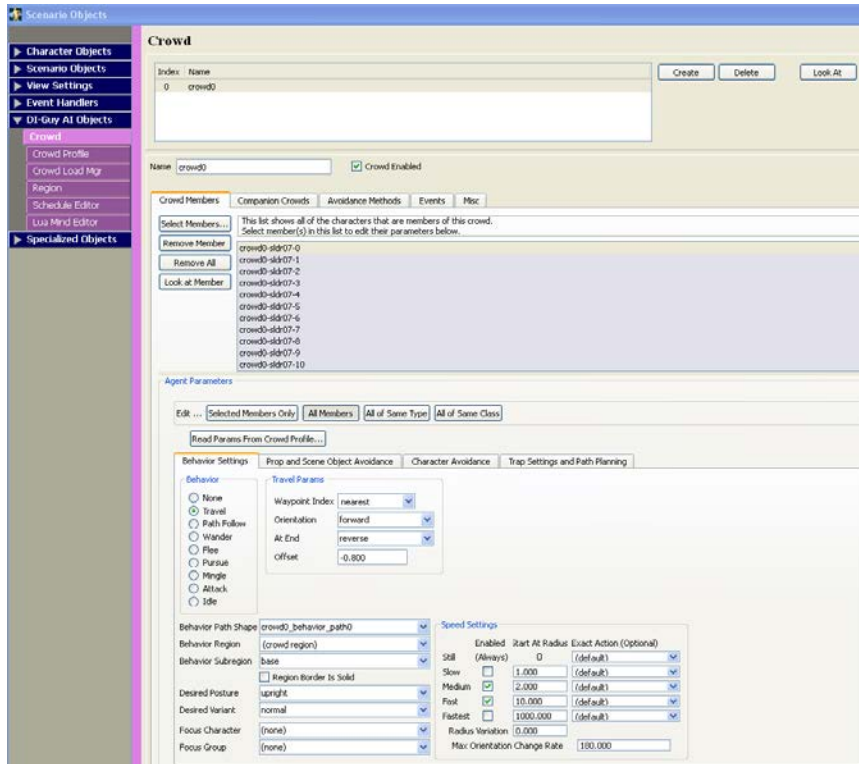
3.9.2 Default Agent Parameters

Please see section 3.11 “Crowd Member Parameters”, for a full description of the default Agent Parameters. This critical user interface is shared by both the Crowd Profile and Crowd tabs, and is therefore described in its own separate section.

3.10 Crowd

The crowd tab shows the parameterized description of each crowd in the scenario. Users may wish to:

- o Edit in more detail an existing crowd before running (Author).
- o Edit an existing Crowd at runtime in reaction to ongoing simulation conditions (SAF Operator).



The Crowd Tab of the AI Elements palette.

Here is a description of the parameters specific to the Crowd Tab:

- **Current Crowd Select Box:** Select the current Crowd for modification. In the image above, “crowd0” is chosen from a list of one profile.
- **Create:** Push the “Create” button to create a new Crowd.
- **Delete:** Push the “Delete” button to delete the selected Crowd.
- **Look At:** Push the “Look At” button to have the camera focus on the Crowd.
- **Name Textbox:** You can alter the name of the current Crowd here.
- **Crowd Enabled:** A checkbox for enabling/disabling the current Crowd.

3.10.1 Crowd Members Tab

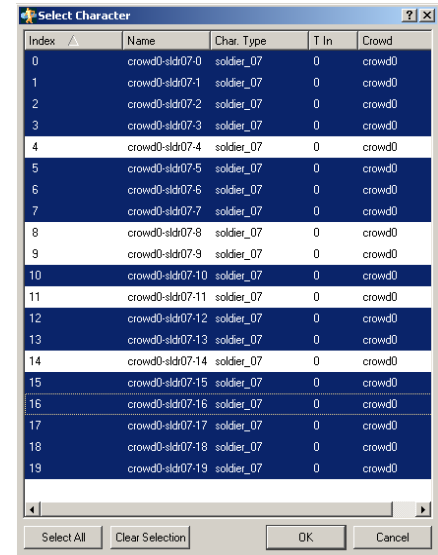
The Crowd Members tab is where most of your crowd editing work will happen. It is here that you can look at and modify the behaviors being used by all, some, or even just a single member of the Crowd.

The Crowd Member Select Box

Near the top, you will find the Crowd Member Select Box. This box lets you review all the Agents that make up your Crowd. You can select a single member and see all the various values associated with its behavior in the Selected Crowd Member Parameters section. You can also add or subtract Agents to the current crowd.

There are also some straightforward buttons associated with the Crowd Member Select Box.

- o Select Members... - This button brings up a dialog box (see figure) which has a list of all the characters currently in the scenario. All characters currently in the crowd will be highlighted in blue. Remember to hold down the <CTRL> key when selecting characters, in order to not deselect other characters. Helper buttons like Select All and Clear Selection are available to assist in selecting members.
- o Remove Member – Remove the highlighted Agent from the Crowd.
- o Remove All – Remove all Agents from the Crowd.
- o Look at Member – Looks at the currently selected agent.



The Select Character dialog box

Selected Crowd Member Parameters Edit Radio-box



Figure -11 - The Selected Crowd Member Parameters Edit Radio-box

This radio-box lets you determine how the Agent parameter edits you make will be applied to the members of the Crowd.

- o Selected Member Only – Only the member selected in the Crowd Member Select Box will have its behavior parameters altered by edits.
- o All Members – Any edit will be applied to all members of the Crowd.
- o All Members of Same Type – Any edit will be applied to members of the class that share the same DI-Guy character type as the member selected in the Crowd Member Select Box.
- o All Members of Same Class – This is an experimental classification that divides DI-Guy characters into major classification classes such as vehicles vs. people.

Read Params from Crowd Profile button

This button takes all the values from a given profile and copies them into the selected Crowd Members current values.

Crowd Member Parameters

Please see section 3.11 Crowd Member Parameters, for a full description of the Crowd Member Parameters section. This critical user interface is shared by both the Crowd Profile and Crowd tabs, and is therefore described in its own separate section.

3.10.2 Companion Crowds Tab



The Companion Crowds Tab

Companion Crowds are Crowds that your current Crowd will do collision avoidance against. DI-Guy AI defaults to all Crowds being Companion Crowds. But it is often useful to remove a Crowd as a Companion Crowd for performance reasons, as of course, checking all members of one Crowd for collision with members of another Crowd can consume significant CPU resources.

In addition to the list of Crowds that are Companion crowds, the following buttons are also used:

- o Selection Companions... - This button brings up a simple dialog box containing all candidate Companion Crowds for selection/de-selection.
- o Remove All – Remove all Companion Crowds.
- o Add All – Add all candidate Crowds as Companion Crowds.

3.10.3 Avoidance Methods Tab

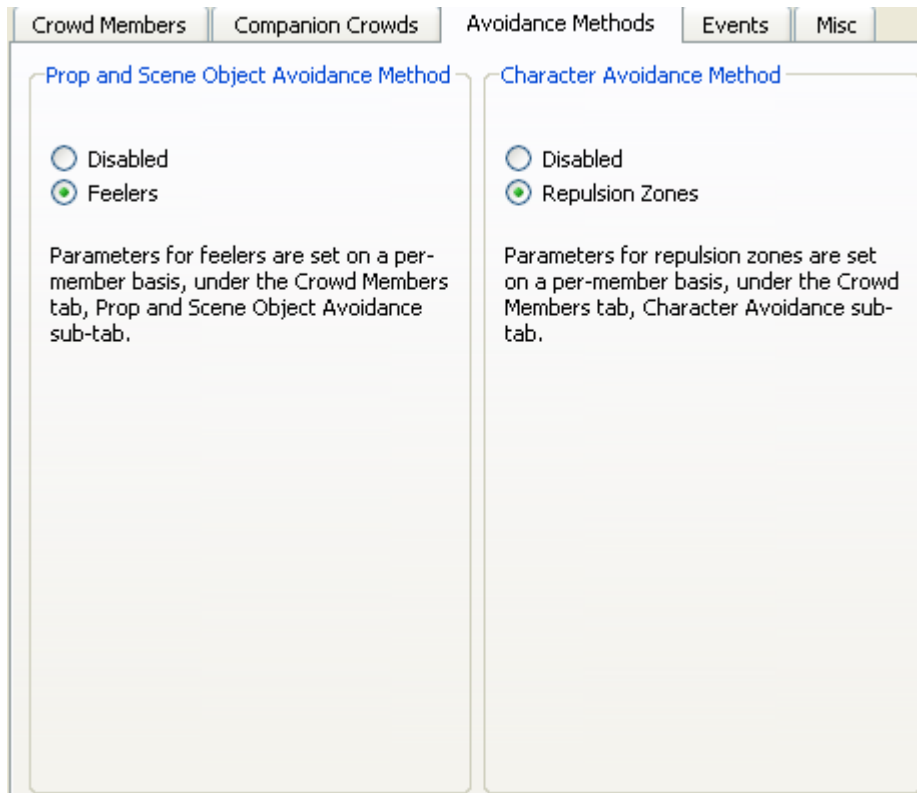
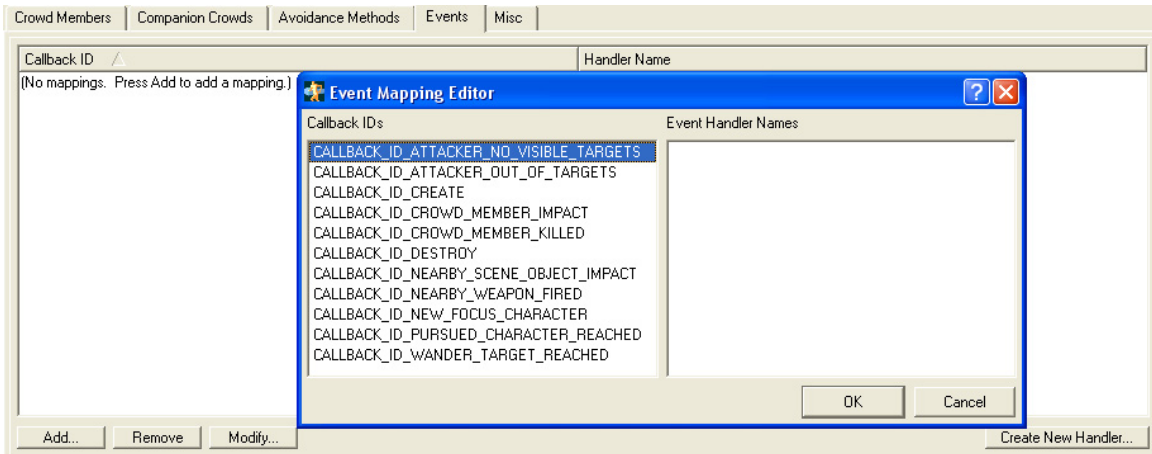


Figure -12 - The Avoidance Methods Tab

The Avoidance Methods Tab is used to enable and disable contact detection methods for both static and dynamic avoidance. This is provided as a method for improving performance.

3.10.4 Events Tab

The Events Sub-Tab is used to hookup various crowd-related events to user-defined event handling scripts or functions. It is recommended that the user review the Event Handlers page as defined in the DI-Guy Scenario User Guide for information on how DI-Guy works with events and event handlers. This section acts very much as a shortcut to that page.



The Events Tab along with the Event Mapping Editor popup

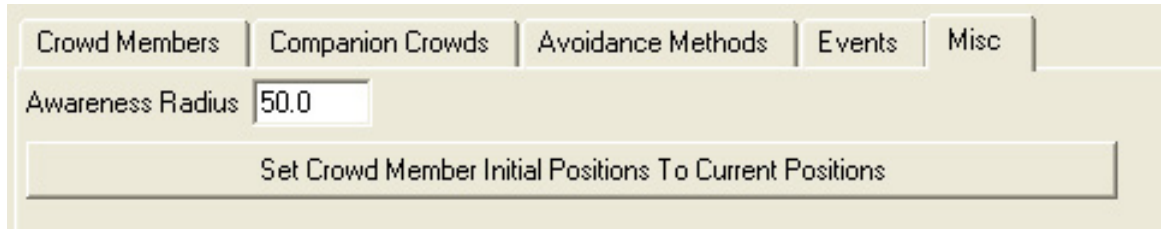
CALLBACKS OF INTEREST: Two special functions, `diguyCrowd::get_callback_character()` and `diguyCrowd::get_callback_impact()`, are provided as part of the `diguyCrowd` API to help facilitate building generic responses to events. Depending on the event, these functions will return the character or impact of interest.

Six `diguyCrowd` class crowd-related events are currently available for event mapping:

- o CREATE – after a Crowd is created, the specified event handler will be called.
- o CROWD MEMBER IMPACT – similar to `diguyCharacter`'s IMPACT event, this event occurs when a detonation occurs near the Agent. Without it, the Agent automatically dies. Useful for adding custom damage models at the crowd level. The callback character is the member of the crowd that received an impact, and the callback impact is the impact that hit the character.
- o CROWD MEMBER KILLED – anytime a member of a Crowd is killed, the specified event handler will be called. The callback character is the member of the crowd that was killed.
- o DESTROY – anytime a Crowd instantiation is destroyed, the specified event handler will be called.
- o NEARBY SCENE OBJECT IMPACT – anytime a detonation occurs within 50 meters of the crowd's current bounds, this event is triggered. Calling `diguyCrowd::get_callback_impact()` should provide the impact information. The callback character is the individual that caused the detonation

- o **NEARBY WEAPON FIRED** – anytime a weapon is fired, either locally or via a FIRE pdu in DIS or HLA equivalent, within an awareness radius (defaults to 50 meters) of the crowd’s current bounds, this event is triggered. The callback character is the individual that caused the detonation.

3.10.5 The Misc Tab



Awareness Radius – This draws an axis-aligned box around the character and will reject all events, such as nearby detonations, that are outside of the awareness radius. NOTE: awareness radius affects ALL members of the crowd, not just the selected crowd member. NOTE: if you want to have different members of the crowd have different awareness levels, use the maximum awareness value here, and then implement the smaller individual awareness levels from your callback function.

Set Crowd Member Initial Position to Current Position button – This is a very useful button for resetting the starting positions of the crowd characters

3.11 Crowd Member Agent Parameters (Crowd and Crowd Profile)

The Crowd Member Parameters section is common to both the Crowd Profile and Crowd tabs. In the former case, the values set there are applied as default values for Crowds when they are blitzed in. In the latter case, the values set there are applied according to the button selected in the Selected Crowd Member Parameters Edit Radiobox.

Crowd Members Parameters is a particularly important part of the GUI for DI-Guy AI, allowing the settings of numerous parameters that affect how Agents behave and navigate.

3.11.1 Behavior Settings Tab

Figure -14 - The Behavior Settings Tab

This very important tab defines the overall behavior of the selected crowd members. It is strongly recommended that you refer to section 3.3 Behaviors for an overview of Behaviors to supplement the information here.

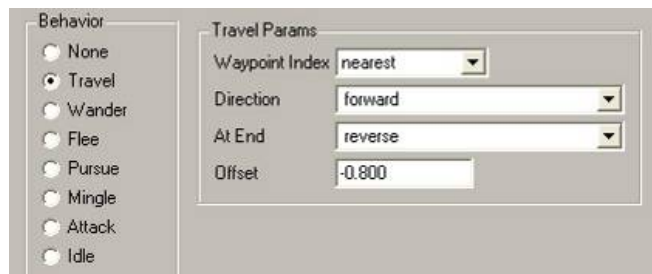
The tab's look changes based on the Behavior selected (see following section), but the following fields are common to all the Behaviors:

- Behavior Path Shape – For behaviors that use a path shape, the particular path shape to use is selected here.
- Behavior Region – The overall region with which the selected crowd member is associated. This rarely needs to be changed; instead the sub-region is more often manipulated.
- Behavior Sub-region – Further refine your behavior with the sub-regions. Note that there is a checkbox here to indicate how rigidly to apply the borders of the region when trying to contain where the Agents are allowed to move.
- Desired Posture – Request a particular posture for the selected crowd member. Typical options are upright, crouched, and/or prone.
- Desired Variant – Request a particular mental/emotional state when the behavior queries the Agent's underlying DI-Guy character action table for a still or locomotive action.
- Focus Character – For actions that can have a focus, the character can be set here.
- Focus Group – Similar to the Focus character, except one can select a group, such as net_friendly, net_neutral, or net_hostile.
- Speed Settings – Define your speed zones and radii here for your Agent, as well as any Exact Action overrides. See section 3.3.1 for descriptions of the Speed Settings and how they work. A radius variation lets you control how uniform the radii are applied. The Max Orientation Change Rate lets you control how quickly a character can turn, which is useful when working with vehicles and animals.

3.11.2 Behavior Settings Descriptions

DI-Guy AI offers nine behaviors. Following is a description of the GUI interface to these behaviors. It is strongly recommended that you refer to section 3.3.1 Behaviors for in depth descriptions of each behavior.

- None – no behavior specified. None is a useful behavior as it allows other behaviors such as those based on paths to be expressed or commanded directly via the DI-Guy API to be applied. When transitioning from another base behavior to none, the character will be commanded to stop.
- Travel – characters will follow a behavior path shape. This is one of the most useful behaviors, and is perfect for when you want to create character “traffic”.



The Travel Behavior

The Behavior Path Shape shows the path shape that the Crowd will use as a topology guide. The Travel Params section lets you select the waypoint index (nearest, random, or a value from 1 to 20), the direction of motion (forward, backward), what to do when you reach the end of the path shape (reverse, loop, start, or teleport), and the allowable offset that characters can drift to the left or right of the path shape.

- o Path Follow – characters will follow a path, performing actions as designated by action beads on that path.

The screenshot shows a configuration window for 'Path Follow Behavior Params'. On the left, a list of behaviors includes 'None', 'Travel', 'Path Follow' (selected), 'Wander', 'Flee', and 'Pursue'. The main panel contains a checked checkbox for 'Pause Path Time Outside Path Behavior', a text input for 'Unpause Path Time At Radius' with the value '4.000', and another text input for 'Position Look-ahead Dist.' with the value '1.000'.

The Path Follow Behavior

The Pause Path checkbox indicates when an agent is put into a new behavior, such as flee, whether he should rejoin the path where he left off, or whether he should rejoin the path where he would have been if had never left the path. The unpause path time indicates when time should readvance, assuming the pause path time was checked. The position look-ahead distance is specified in meters.

- o Wander – characters perform their still action, then choose another random point in the region, move to that new point while avoiding other characters and obstacles, and then perform the still action again. Wander is one of the most fundamental behaviors in the DI-Guy AI arsenal.

The screenshot shows a configuration window for 'Wander/Mingle Params'. On the left, a list of behaviors includes 'None', 'Travel', 'Wander' (selected), 'Flee', 'Pursue', and 'Mingle'. The main panel contains text inputs for 'Move On Interval Min' (5.000), 'Move On Interval Max' (10.000), and 'Max Dist To Wander' (8.000). It also features a checked checkbox for 'Allow Desired Orientation Change' and an unchecked checkbox for 'Turning Actions Enabled'.

The Wander Behavior

In the Wander/Mingle Params, the interval min and max settings indicate the number of seconds that the character should perform the still action before picking a new point in the region. Checkboxes allow the user to allow orientation changes and/or turning actions. The Max Dist to wander allows a limit to be set to how far away a character will move before again performing a still action.

- o Flee – Characters run away from a particular point, character or group. This is typically used when fleeing from explosions, enemy characters, or other dangers. To trigger the flee behavior from the API, use `diguyCharacter::agent_flee_(character/location/group)`.

- o Pursue – Characters follow the focus character or characters from the focus group

The screenshot shows a GUI for configuring AI behavior. On the left, a vertical list of behavior options includes 'None', 'Travel', 'Path Follow', 'Wander', 'Flee', 'Pursue' (which is selected with a green dot), 'Mingle', 'Attack', and 'Idle'. To the right, the 'Attack/Pursue Params' section contains several input fields: 'Max Focus Distance' (100.000), 'Attack Distance' (10.000), 'Attack Fire Interval' (0.200), and 'Fire Interval Variation' (0.050). There is also an unchecked checkbox for 'Attack Stands Ground'. Further right, the 'Speed Match' section has a checked checkbox for 'Enable Pursue Speed Match', an 'Update Interval' field (2.000), and a 'Max Speed Up Factor' field (2.000).

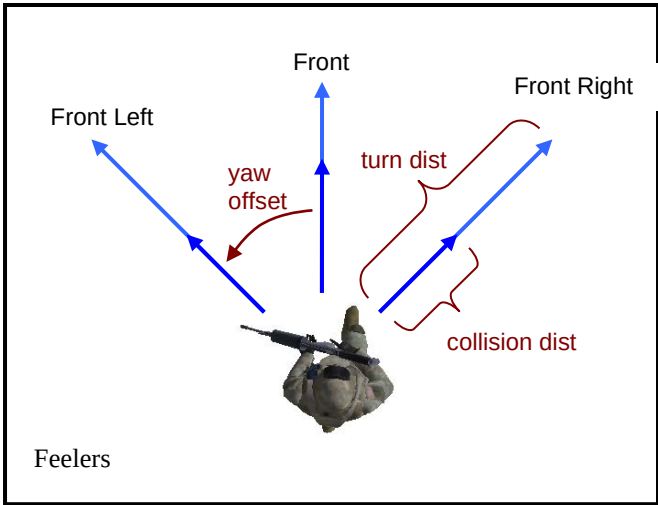
Figure -18 - The Pursue Behavior

The Max focus distance instructs the Agent not to pursue if the focus character is greater than some number of meters away, while the Attack distance specifies when the pursue character believes it has achieved its goal of reaching the character. Attack Fire interval and Fire Interval Variation is not used by Pursue. Enable Pursue Speed Match enables characters to set their speed to exactly the same speed as the focus character they are following. This is useful when you are trying to have a line of different sized characters following each other. The matching speed test is performed every Speed Match Update Interval and the max speed up factor provides a limit to how much speed up can be applied.

- o Mingle – this is a behavior similar to Wander, but characters will look for other agents in the crowd that are also mingling, and choose their next random point to be near one of those characters, and make that character be their new focus character. The behavior also has a concept of “claustrophobia”, so that when things get too crowded, the character will instead pick a point far from other characters. The parameters in the GUI are identical to Wander.
- o Attack – built on top of pursue, characters that are attacking will fire their weapon at the focus character or group if they are able to aim a weapon. Note that the Attack Distance is now used to indicate when the Agent will stop and fire its weapon. The Attack Fire Interval indicates the number of seconds between shots fired.
- o Idle – With the exception of “None”, all of the other base behaviors handle the setting of the desired position of the character. “Idle” does not. This makes “Idle” very useful when you want to create a character where you set the desired position of the character, but still retain all the AI capabilities of the character. “Idle” can be used as the base behavior when making a sophisticated character with an AI Mind.

3.11.3 Prop and Scene Object Avoidance

Agents use *feelers* to prevent collision with other static objects in the scene. (Note: Path planning is an alternate method)



This Static Avoidance tab allows the feelers parameters to be programmed.

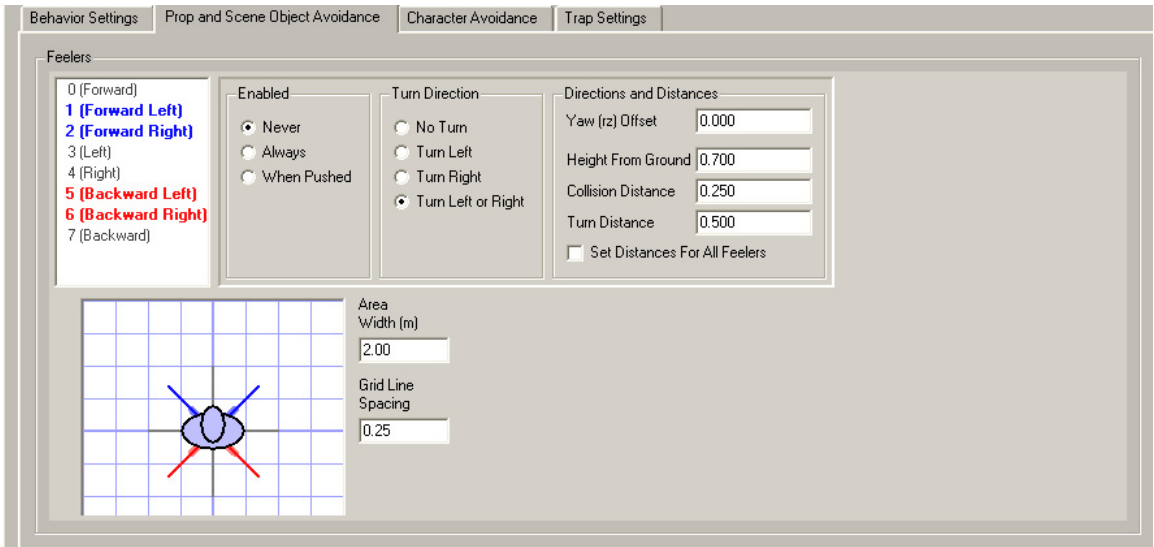


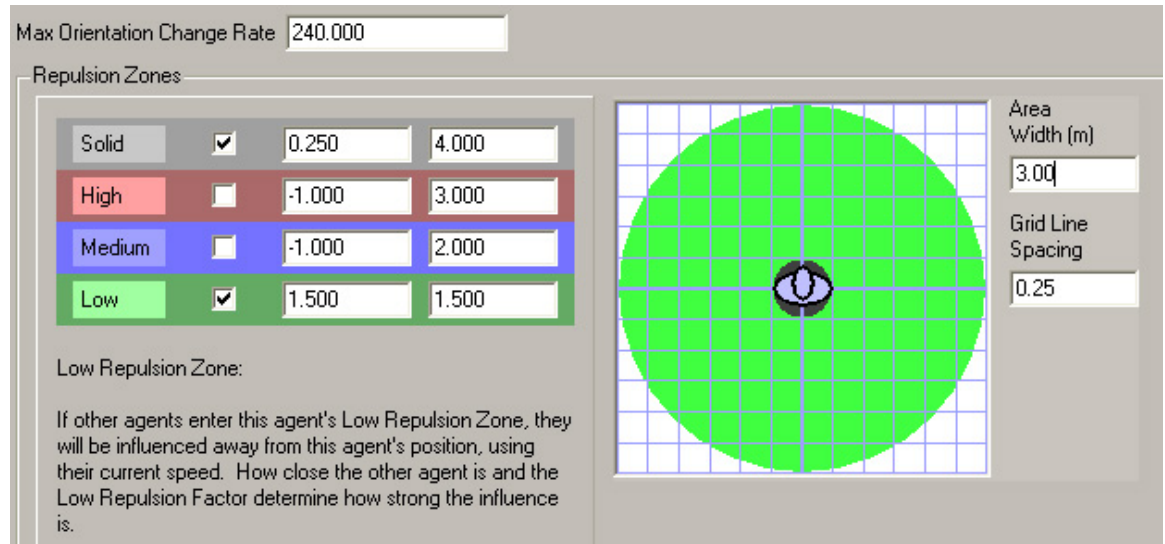
Figure -21 - The Static Avoidance Tab

Up to eight feelers can be specified. When a feeler is highlighted in the leftmost box, the right 3 boxes correspond to its parameterizations. Most important (and basic) is the radio-box that indicates whether the feeler is enabled or not, or only enabled when the character has been “pushed” due to avoidance/repulsion of other characters or static objects. The turn direction indicates what way to turn the character when a collision to that feeler is detected. The Yaw Offset indicates the angle (0 is forward, 90 is pointing left,

etc), The Height From Ground specifies the height of the feeler and its length. The Collision Distance and Turn Distance specify when the character needs to turn and when he needs to take more evasive action to avoid a collision. Please consult section 3.6.2 for a thorough description of how the feelers work.

3.11.4 Character (Dynamic Object) Avoidance

Agents use feelers to prevent collision with other dynamic objects in the scene too. Dynamic objects are other agents from the current Crowd or from a Companion Crowd.



Max Orientation Change Rate: 240.000

Repulsion Zones:

Zone	Factor	Radius (m)	Area Width (m)
Solid	☒	0.250	4.000
High	☐	-1.000	3.000
Medium	☐	-1.000	2.000
Low	☒	1.500	1.500

Grid Line Spacing: 0.25

Area Width (m): 3.00

Low Repulsion Zone:

If other agents enter this agent's Low Repulsion Zone, they will be influenced away from this agent's position, using their current speed. How close the other agent is and the Low Repulsion Factor determine how strong the influence is.

The Dynamic Avoidance Tab

When avoiding another dynamic character, the Max Orientation Change Rate limits how much the character can turn to avoid that character. The angle is specified in degrees.

Characters also specify four radii of decreasing size (low, medium, high, and personal space) to determine how dramatically to attempt to avoid collision with the other dynamic object. Along with an associated “repulsion factor”, characters will select how dramatically to yaw their current orientation (and typically their direction of travel), as well as how fast to travel, to avoid collision. See section 3.6.4 for a description of dynamic object avoidance. In the GUI above, the character starts repelling characters when they are within 1.5 meters with a “force” of 1.5 and repel more strongly with a “force” of 4 when the characters are within 0.25 meters of each other.

The Area width and grid line spacing are simply values to set and understand graphically your chosen repulsion zone configuration.

Influence Center Shift

This allows you to set the point in your character that is the center of the repulsion zones. This is typically 0,0,0.

3.11.5 Trap Settings and Path Planning

A constant challenge for autonomously controlled characters is being able to successfully navigate through terrain of which they may have limited knowledge. A typical failure mode is that a character will get caught in an endless loop of cycling between 2 or more obstacles it is avoiding. This tab is devoted towards parameters that let you instrument DI-Guy AI to minimize or eliminate your Agents getting caught in corners. Please consult section 3.7 “Detecting and Getting out of Traps” section for a thorough discussion of the Traps concept.

Behavior Settings				Prop and Scene Object Avoidance		Character Avoidance		Trap Settings and Path Planning	
Untrap Methods									
Wedged Trap	Time Threshold	0.800	Untrap Method	random_turn					
First No Progress Trap	15.000	ghost							
Second No Progress Trap	30.000	teleport							
Random Turn Method Params									
Max Tries	4								
Ghost Method Params									
Ghost Duration	1.000								
Nav Path and Path Planner Params									
Preferred Subregions					Repulsed Subregions				
☐ populate	☐ populate								
☐ red	☐ red								
☐ green	☐ green								
☐ blue	☐ blue								
☐ yellow	☐ yellow								
☐ orange	☐ orange								
☐ white	☐ white								
☐ road	☐ road								
☐ sidewalk	☐ sidewalk								
☐ crosswalk	☐ crosswalk								
Preferred Cost Bias		0.250							
Neutral Cost Bias		1.100							
Repulsed Cost Bias		10.000							
Fallback Region		navmesh							
☒ Wander Uses Path Planner									
☒ Pursue Uses Path Planner									
Max LOS Travel Dist		1000.000							

The Trap Settings Tab

Untrap Methods lets you specify how the Agent will handle trap conditions. There is a Time Threshold in seconds coupled with a pulldown for specifying the Untrap Method desired.

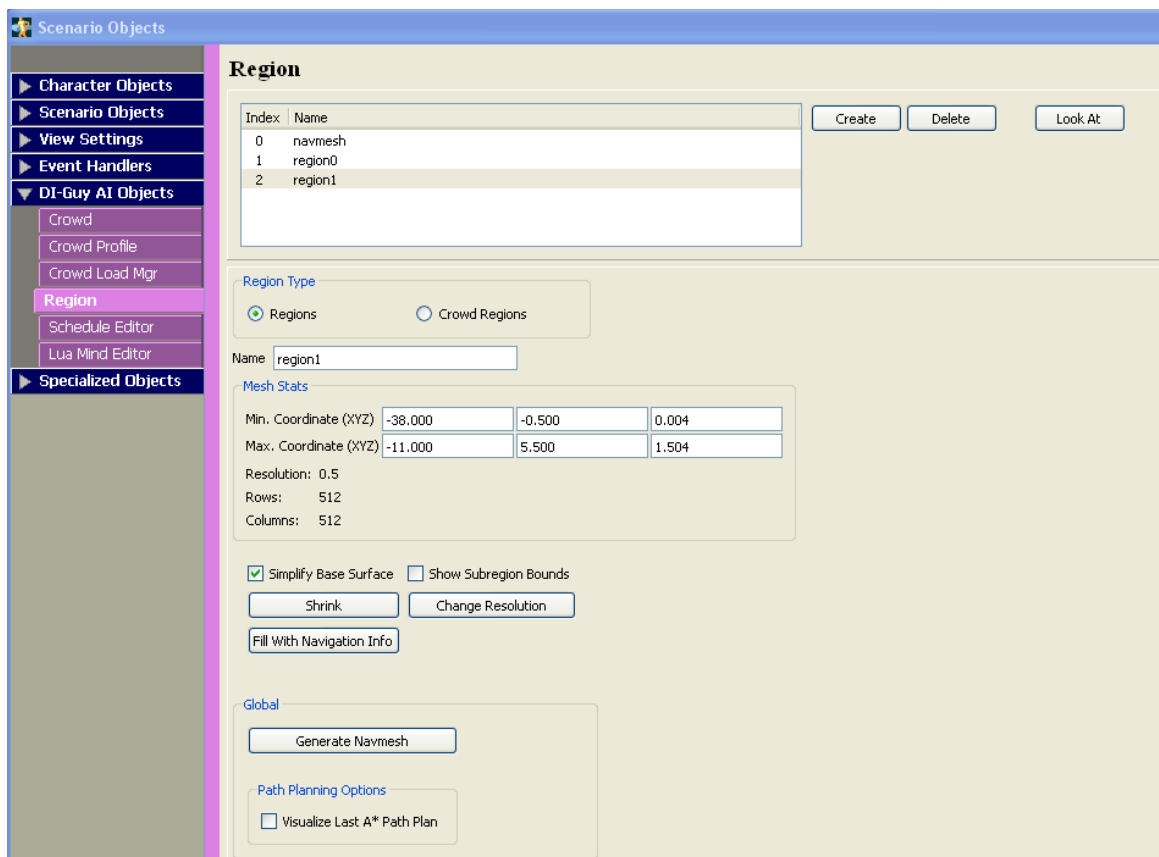
Choices for Untrap methods include:

- o None – no algorithm is applied to avoid traps.
- o Nav_path – when a trap is detected, use the Navigation Path to find a navigable path to the desired destination. Navigation paths are created via an A* Algorithm (http://en.wikipedia.org/wiki/A*_search_algorithm). As such, they require a valid navmesh region to navigate. Unpainted areas are treated as impassable. As long as valid data is

provided, the navmesh is guaranteed to return the shortest path. Navmesh does not navigate between noncontiguous regions.

- o Random_turn – choose a random amount to yaw the character in an attempt to escape the trap by trial and error. Note that there is an associated parameter for setting the maximum number of times a random turn will be employed before advancing to the next untrap method.
- o Ghost – allow the character to temporarily pass through a wall or other obstacle in an attempt to escape the trap. Note that there is an associated parameter displayed to set how long the character will turn off collision detection when ghosting through the wall.
- o Teleport – the character will teleport (change its position discontinuously) to evade the trap.
- o Stop moving – switch to the still action and make no attempt to escape the trap.

3.12 The Region Page



The Region Page

The Region tab of the AI Elements palette is where you can create and review Regions used in coordination with the Crowds and Behaviors. Note that the Input palette's Region Painter Mode also allows you to manipulate Regions.

Here is a description of the parameters specific to the Region Tab:

- Current Region Select Box: Select the current Crowd for modification. In the image above, “region1” is chosen from a list of one region.
- Create: Push the “Create” button to create a new Region.
- Delete: Push the “Delete” button to delete the selected Region.
- Look At: Push the “Look At” button to have the camera focus on the Region.
- Region Type: There are two types of regions. Crowd Regions are regions associated with a specific crowd and plain Regions are more global in nature.
- Name Textbox: You can alter the name of the current Region here.
- Mesh Stats: Review the mesh statistics in order to manage complexity vs. performance
 - Min/Max Coordinates – values of the extents of the Region.
 - Resolution – mesh points per meter (read only).
 - Rows – number of rows in the mesh (read only).
 - Columns – number of columns in the mesh (read only).
- Shrink – shrink the Region (useful for efficiency, will fit region to the extents of the sub-regions that compose it).
- Simplify Base Surface – documentation pending
- Show subregion bounds – documentation pending
- Fill with navigation info – documentation pending
- Force the generation of a navmesh region, which will be used by agents for smart navigation via A Star Path Planning. Note that you should not have a “navmesh” region when you click on this, instead, the “navmesh” region will be created as output.
- Path Planning Options – visualize last AStar Path Plan – graphically display in the 3D Window the most recently generated path as provided by the navmesh and AStar algorithm.

3.13 Schedule Editor

The schedule editor lets the author create “patterns of life” for their agents by giving them goals for certain parts of the day. The user interface is fairly self-explanatory. While the scenario can have multiple schedules, only one is active for editing. Controls for creating, deleting, and creating copies are located at the top of the user interface.

Schedule Editor

Active Schedule:

Name:

Schedule Events

Time	Action	Role	Action	Percentage	Time Window	Group
12:00	Trigger Signal	luaMiddleEasternPedestrian	get hungry	20	1	all
12:01	Trigger Signal	luaMiddleEasternPedestrian	want company	30	1	>>
12:02	Trigger Signal	luaMiddleEasternPedestrian	get angry	20	1	>>
12:03	Trigger Signal	luaMiddleEasternPedestrian	call to prayer	30	1	>>
12:04	Trigger Signal	luaMiddleEasternPedestrian	get happy	40	1	>>
12:05	Trigger Signal	luaMiddleEasternPedestrian	get hungry	10	1	>>
12:06	Trigger Signal	luaMiddleEasternPedestrian	want company	20	1	>>
12:07	Trigger Signal	luaMiddleEasternPedestrian	get hungry	30	1	>>
12:08	Trigger Signal	luaMiddleEasternPedestrian	get happy	20	1	>>
12:09	Trigger Signal	luaMiddleEasternPedestrian	call to prayer	30	1	>>

The heart of the user interface is the schedule editor in the bottom half of the page. Here you can append, insert, and delete entries (one per row) for “Actions” that occur during the day. The fields for each entry are:

- o Active checkbox: events that are not checked are disabled.
- o Time: time that event should occur
- o Action: Trigger Signal (send a signal to Agents with the mind subclass shown in the next field)
- o Role: all agents with the Mind class specified in the Role field and also are satisfy the Group designation field are candidates to receive the Action specified in the previous field.
- o Action: For Trigger Signal, the name of the Action.
- o Percentage: Proportion of candidate Agents to receive the Action who will be sent the Action.
- o Time Window: Range in minutes to “spread out” the actions so that all the Agents receiving an Action don’t get them at the exact same moment. Used to increase realism.
- o Group: Along with Role, this field designates which Agents are candidates to receive the scheduled action. Possible groups include “all”, “net friendly”, “net neutral”, “net opposing”, and “reflected crowd human”.

4. Visibility – crowd and agent-related settings

Scenario Objects

- ▶ Character Objects
- ▶ Scenario Objects
- ▼ View Settings
 - Camera
 - Fog
 - Light
 - Visibility**
- ▶ Event Handlers
- ▶ DI-Guy AI Objects
- ▶ Specialized Objects

Visibility

Characters

Characters	Character Labels	Aim Trajectories
☐ None	☐ None	☒ None
☐ Current	☐ Current	☐ Current
☒ All	☐ All	☐ All
	☒ Per Object	

Paths

Path Shapes	Waypoints	On-Path Beads	Action Labels
☐ None	☐ None	☐ None	☒ None
☐ Current	☒ Current	☒ Current	☐ Current
☐ All	☐ All	☐ All	☐ All
☒ All of Current			

Regions

Sensor Regions	Regions	Crowd Regions
☐ None	☐ None	☐ None
☐ Current	☒ Current	☐ Current
☒ All	☐ All	☒ All

Crowds and AI

Member Feelers	Member Influence Area	Member Behavior Visuals	Lua Objects
☒ None	☒ None	☒ None	☐ None
☐ Current Crowd	☐ Current Crowd	☐ Current Crowd	☐ Current
☐ All	☐ All	☐ All	☒ All
			☐ Per Object

Particle Systems

☐ Hide Emitter Box
☒ Show Emitter Box

Lights

☒ None
☐ Current
☐ All

Frills

☒ Shadow Disks

Appearance Effects

☒ Enable Appearance Effects
☐ Disable Net-Generated Effects

Figure -25 - The Visibility Tab of the Scenario Objects palette.

Path Shapes – important when working with the Travel Behavior – Toggle visibility of path shapes.

Regions – Toggle global and crowd region graphic on and off for the current crowd or all crowds.



Figure -26 - The optional crowd region graphic is shown for a crowd of world famous DI-Guy chickens. Note that the dark points indicate the underlying mesh points, while the red lines indicate the extents of the region.

Crowd Member Feelers – Toggle the member feelers on and off for the current crowd or for all crowds.

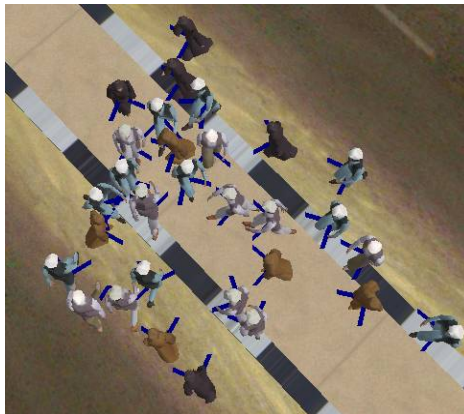


Figure -26 - Feeler graphics enabled for a dense crowd of civilians.

Crowd Member Influence Area – toggle the crowd member influence area graphic on and off for the current crowd or all crowds. This graphic shows repulsion radii for the characters.



Figure -27 - Crowd member Influence Graphics

Member Behavior Visuals - toggle the crowd member behavior visual graphic on and off for the current crowd or all crowds.

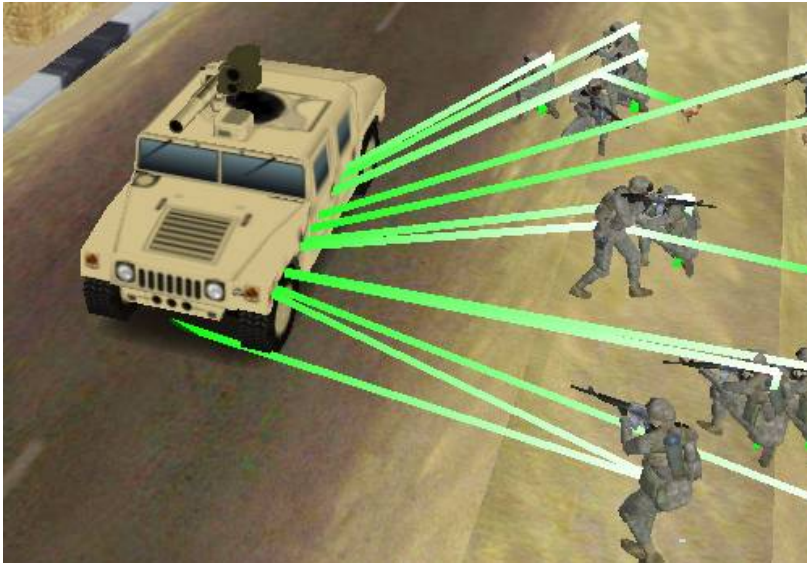


Figure -28- A group of soldiers in Wander mode have the humvee as their focus character and kneel_aim as their still action.

Lua Objects – Documentation pending

4.1 View Settings – Camera Tab – crowd-related settings

The Symbolic View toggle on the camera tab of the View Settings palette allows the display of symbols instead of graphics for the entities. This is often a popular view settings for SAF operators.

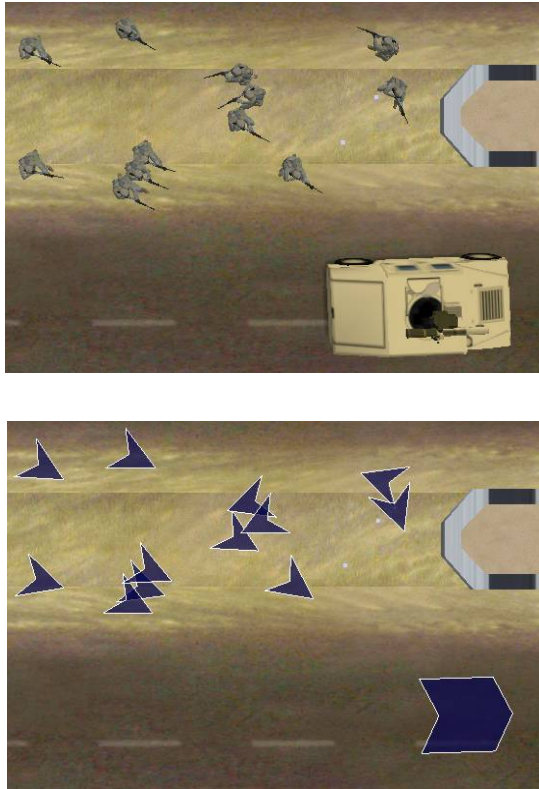


Figure -30 – A Crowd before and after Crowd member Influence Graphics

Path Planning and Navigation Meshes

DI-Guy AI characters can optionally use a Navigation Mesh to move about the terrain and avoid local minima and traps where they might get stuck. Path planning using the navmesh is an attractive alternative to using the feelers to navigate. There are two ways that the path planner may be triggered; automatically during base behaviors, and explicitly via the API.

4.2 Automatic Path Planning

In wander and pursue, when a character is attempting to navigate the world and can't see its target location it will attempt to path plan to the location. This functionality can be controlled via the diguyAgentParams class or via the Trap Settings and Path Planning UI:

The screenshot shows the 'Agent Parameters' window with the 'Trap Settings and Path Planning' tab selected. The interface includes several sections for configuring agent behavior:

- Buttons:** 'Edit ...', 'Selected Members Only', 'All Members', 'All of Same Type', 'All of Same Class', and 'Read Params From Crowd Profile...'.
- Behavior Settings:** Includes 'Untrap Methods' with a table:

	Time Threshold	Untrap Method
Wedged Trap	0.800	random_turn
First No Progress Trap	15.000	ghost
Second No Progress Trap	30.000	teleport
- Random Turn Method Params:** 'Max Tries' set to 4.
- Ghost Method Params:** 'Ghost Duration' set to 1.000.
- Nav Path and Path Planner Params:**
 - Preferred Subregions:** A list of checkboxes for 'populate', 'red', 'green', 'blue', 'yellow', 'orange', 'white', 'road', 'sidewalk', and 'crosswalk'.
 - Repulsed Subregions:** A similar list of checkboxes for the same categories.
 - Neutral Cost Bias:** Set to 12.000.
 - Repulsed Cost Bias:** Set to 100.000.
 - Fallback Region:** A dropdown menu currently set to 'navmesh'.
 - Wander Uses Path Planner:** Checked.
 - Pursue Uses Path Planner:** Checked.
 - Max LOS Travel Dist:** Set to 1000.000.

Agents will attempt to factor in the Preferred and Repulsed parameters when they perform a path plan. Being able to define preferred and repulsed sub-regions lets you do things like encourage pedestrians to use crosswalks and sidewalks and avoid streets. By default they will use their crowd region for creating a path, but a fallback region can also be specified.

4.3 Path Planning via the API

The most direct way to use the navigation mesh is to call the “move to” functions. The following functions invoke path planning when executing:

- diguyCharacter::agent_move_to_point()
- diguyCharacter::agent_move_to_point_bg()
- diguyCharacter::agent_move_to_point_via_subregions()
- diguyCharacter::agent_move_to_point_via_subregions_bg()
- diguyCharacter::agent_move_to_region()
- diguyCharacter::agent_move_to_region_via_subregion()
- diguyScenario::find_navigation_path()

Defining a Navigation Mesh

Any region in DI-Guy can be used as a navigation mesh. Typically, however, we recommend defining a region called “navmesh” and having your code reference that.

The screenshot shows the 'Scenario Objects' window with the 'Region' panel selected. The panel contains a table of regions, a 'Region Type' section, a 'Name' field, 'Mesh Stats', and various configuration options.

Index	Name
0	navmesh
1	region0
2	region1

Buttons: Create, Delete, Look At

Region Type: ☒ Regions ☐ Crowd Regions

Name:

Mesh Stats:

Min. Coordinate (XYZ)		
-38.000	-0.500	0.004
Max. Coordinate (XYZ)	-11.000	5.500
	1.504	

Resolution: 0.5
Rows: 512
Columns: 512

☒ Simplify Base Surface ☐ Show Subregion Bounds

Buttons: Shrink, Change Resolution, Fill With Navigation Info

Global:

Path Planning Options: ☐ Visualize Last A* Path Plan

The recommended approach is to generate your region (mesh) by using the Generate Navmesh button in the Region page. Alternatively, the region can be painted using RegionPainter or calling the function `diguyRegion::generate_navmesh()`.

Editing a Navigation Mesh

Check your navmesh for holes or areas where it may unintentionally go through a wall. Use the visualizations for regions to aid with this monitoring. Use the painting and erasing brushes to edit your navigation mesh to fix any anomalies.

Using the navigation mesh with the base behaviors

Navmesh is an option in the trap settings for crowds. The system will typically use the current region as the navigation mesh.

Using preferred and repulsed subregions when navigating

Being able to define preferred and repulsed subregions lets you do things like encourage pedestrians to use crosswalks and sidewalks and avoid streets. Consider the following navigation function:

```
int agent_move_to_point_via_subregions(float x, float y, float z,
    const char* via_region = NULL,
    int preferred_subregions_mask = DIGUY_SUBREGION_BASE_MASK,
    float cost_bias_for_neutral_regions = 12.5f,
    int repulsed_regions_mask = DIGUY_SUBREGION_NONE_MASK,
    float cost_bias_for_repulsed_regions = 100.0f);
```

Description:

Agent will attempt to create a navigation path on the `via_region` and then travel it. The `subregion_mask` is used to specify what preferred subregions to use. An A* path planning algorithm is used to find the path. A cost bias value can be used to make non-preferred spaces still navigable.

Arguments:

<code>x, y, z</code>	the target location
<code>via_region</code>	the name of the region to run A* on
<code>preferred_subregions_mask</code>	a number of diguySubregionMask or'ed together e.g. DIGUY_SUBREGION_SIDEWALK_MASK DIGUY_SUBREGION_CROSSWALK_MASK
<code>cost_bias_for_neutral_regions</code>	How much more expensive it will be to cross spaces that are not part of desired subregion. Lowers likelihood that search fails on disjointed subregions, Pass

`repulsed_regions_mask` DIGUY_DEFAULT_FLOAT to restrict to just the subregion_mask. This should be a value > 1.0.
 a number of [diguySubregionMask](#) or'ed together e.g. DIGUY_SUBREGION_SIDEWALK_MASK | DIGUY_SUBREGION_CROSSWALK_MASK

`cost_bias_for_repulsed_regions` How much more expensive it will be to cross spaces that are marked as repulsive, Pass DIGUY_DEFAULT_FLOAT to avoid completely.

Returns:

0 on success, -1 on failure

Here is a small code snippet showing an example use of this functionality:

```
-- Retrieve a target xyz coordinate from a subregion
region:get_point_in_region_near_point(DIGUY_SUBREGION_BLUE, x, y, z, 5);

-- Designate the sidewalk and crosswalk subregions in a bitmask
local search_flags = bit.bor(DIGUY_SUBREGION_MASK_CROSSWALK,
DIGUY_SUBREGION_MASK_SIDEWALK, DIGUY_SUBREGION_MASK_BLUE);

-- Move to the xyz location, preferring the crosswalks and sidewalks with a
cost bias of 12:1
luaChar.character:agent_move_to_point_via_subregions(x, y, z, "navmesh",
search_flags, 12);
```

When a character arrives at their target location a `diguyCharacter::CALLBACK_ID_AGENT_TRAVEL_FORWARD_DEST_REACHED` callback is triggered. This functionality is frequently used by minds to path plan to a location and then sleep until arrival.

Note: the above script uses the `agent_move_to_point_via_subregions()` method which can cause high CPU usage and therefore noticeable frame-rate irregularities should the path required be long or difficult to compute, especially if multiple characters perform the same script at roughly the same time. As an alternative, there is a threaded background version of this planning method (called `agent_move_to_point_via_subregions_bg()`) that is described in the next section (Section 3.4).

4.4 Background Path Planning

The algorithms that DI-Guy uses to perform path planning can occasionally consume enough CPU to noticeably affect the playback frame rate. This is true when the path is complicated or when there are several characters path planning at the same time. Two methods introduced in DI-Guy 12 help alleviate the possibility of frame-rate hiccups, but they can require extra code.

- diguyCharacter::agent_move_to_point_bg()
- diguyCharacter::agent_move_to_point_via_subregions_bg()

These two methods have the same arguments as their non-background version. They return immediately to the caller and either has: 1) encountered an error, 2) finished planning, or 3) decided that the planning is complicated enough to warrant moving it into a background thread. In the latter case, the caller may choose to wait on the thread (waiting will not slow down the frame rate) to see whether or not it was successful at computing a path plan. This background planning will result in either 1) success or 2) failure to produce the plan. If you are updating an existing scenario that didn't check for path planning errors or the handle the processing of messages en-route, you can simply add the "_bg" trailer to your existing scenario to upgrade it.

However, if the scenario needs to handle path planning errors, then it has to wait for the planning to complete. If planning is pushed into the background, you can call `wait_for_message()` or `sleep_process_messages()`, but other messages may continue to come in. It can complicate the logic if you wish to handle those messages en-route. Below is an example of a DI-Guy 11 (foreground) vs. a DI-Guy 12 (background) script that handles incoming messages while moving to the desired point (x,y,z). Both have error handling in case the path-planning fails that also complicates the logic and may or may not be needed in your scenario.

■ (FOREGROUND VERSION ... May affect the frame-rate)

```
-- plan a path and move to the point, or return "restart" on failure

local res = luaChar.character:agent_move_to_point_via_subregions(x, y, z,
    "navmesh", search_flags, .25, 1.2)

if (res ~= 0) then return "restart" end

-- wait until we get there, but process messages en-route

luaChar.character:add_wakeup_callback(
    diguyCharacter_CALLBACK_ID_AGENT_TRAVEL_FORWARD_DEST_REACHED)

local ret = luaChar:sleep_process_messages(25,
    luaChar.process_messages_enroute)

luaChar.character:remove_wakeup_callback(
    diguyCharacter_CALLBACK_ID_AGENT_TRAVEL_FORWARD_DEST_REACHED)

-- standard interrupt checking

if luaChar:is_new_state(ret) then
    return luaChar:invoke_state(ret)
elseif ret == "restart" then
    return "restart"
end
```

- (BACKGROUND VERSION ... Shouldn't affect the frame-rate)

```
-- plan a path and move to the point, or return "restart" on failure
local res = luaChar.character:agent_move_to_point_via_subregions_bg(
    x, y, z, "navmesh", search_flags, .25, 1.2)

-- see if the planning happened, or failed, or was pushed into the background
if (res <= -1) or (res == DIGUY_NAV_PATH_ERROR_OUT_OF_TIME) then
    return "restart"
elseif (res == DIGUY_NAV_PATH_ERROR_SEARCH_QUEUED) then

    -- planning is queued, so we wait until we get an answer, but we also service
    -- a message handling function. We want "plan completed" to happen

    luaChar.character:add_wakeup_callback(
        diguyCharacter_CALLBACK_ID_AGENT_PATH_PLAN_COMPLETED);

    res = luaChar:wait_for_message("CALLBACK_ID_AGENT_PATH_PLAN_COMPLETED",
        luaChar.process_messages_enroute);

    luaChar.character:remove_wakeup_callback(
        diguyCharacter_CALLBACK_ID_AGENT_PATH_PLAN_COMPLETED);

    -- if we get a 0 back when waiting, our message arrived
    if (res == 0) then
        local path_plan_res = luaChar.character:get_path_planning_result()
        if ( path_plan_res <= -1 ) or (path_plan_res == 1) then
            return "restart"; -- error occurred, or we timed out
        end
    else
        -- standard interrupt checking
        if luaChar:is_new_state(res) then
            return luaChar:invoke_state(res)
        elseif (res == "restart") then
            return "restart"
        end
        -- other messages?
    end
end

-- wait until we get there, but process messages enroute

luaChar.character:add_wakeup_callback(
    diguyCharacter_CALLBACK_ID_AGENT_TRAVEL_FORWARD_DEST_REACHED)

local ret = luaChar:sleep_process_messages(25,
```

```

    luaChar.process_messages_enroute)

luaChar.character:remove_wakeup_callback(
    diguyCharacter_CALLBACK_ID_AGENT_TRAVEL_FORWARD_DEST_REACHED)

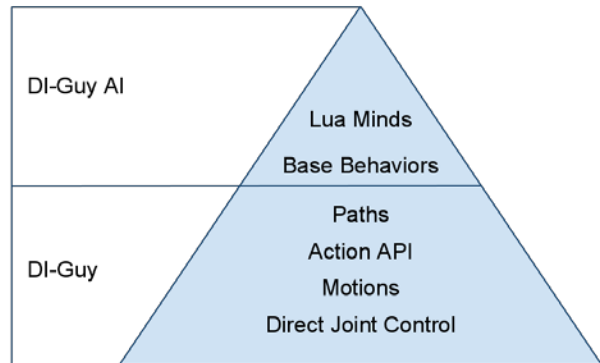
-- standard interrupt checking

if luaChar:is_new_state(ret) then
    return luaChar:invoke_state(ret)
elseif ret == "restart" then
    return "restart"
end

```

5. DI-Guy AI Minds

AI Minds are at the top of the Behavior pyramid. A Mind is a set of lua-based classes that provide artificial intelligence to a character for governing its behavior; most often, the Mind is modeled as a Hierarchical Finite State Machine (HSFM). Each individually state typically expresses the state visually via an AI Base Behavior, Path, or direct commands to the DI-Guy API for any combination of Actions/Gestures/Expressive Faces, etc. State changes, as we will see, can be caused by changes in internal state but are most often the result of incoming messages and events from other characters, ballistic events, and SmartObjects in the scenario.



USING DI-GUY AI MIND PROGRAMMING TIP: Don't get hung up trying to understand every nuance of a DI-Guy AI Mind at first – and definitely don't start by trying to create your own AI Mind from scratch! Instead, pick the DI-Guy AI Mind class closest to what your end character will be, such as `luaSimpleSoldier` or `luaPedestrian`, and start using it. Then, assuming you want to code and customize, create your own subclass from that mind. You'll be able to quickly augment the behavior – and still get all the great out-of-the-box functionality and behavior that attracted you to this solution in the first place!

5.1 The DI-Guy Lua Documentation Files

DI-Guy AI comes with a built-in but extensible set of stock Minds that users are encouraged to use as-is or as starting points for creating even more intelligent characters. In addition to this document, users are

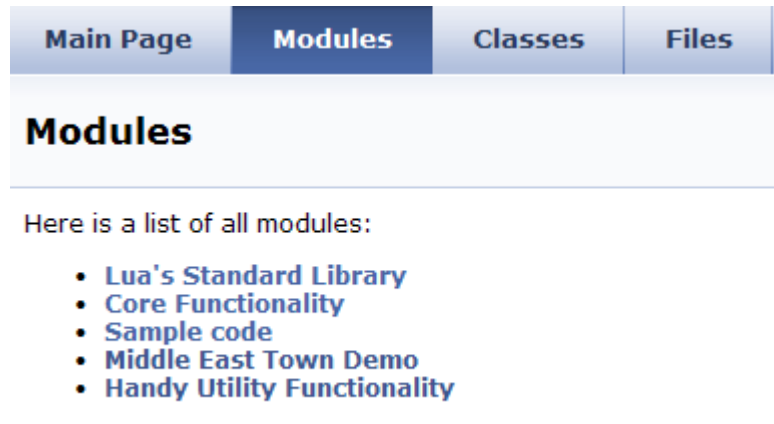
encouraged to consult the DI-Guy: Lua Documentation in the file:

%DIGUY%\doc\diguy_sdk_reference\lua\index.html.

The DI-Guy Lua documentation is broken into three sections:

5.1.1 DI-Guy Lua Modules

Because DI-Guy's lua solutions are Open Source, the user is free to peruse our implementations to get ideas and templates for how to extend the code to fit their needs. The modules section includes 5 entries:



The Standard Library section provides documentation on classes that support the AI Minds but are not directly related to human simulation or AI, such as table, debugging, and math classes.

The Core Functionality documents critical classes such as luaCharacter and luaSmartProp. These are the building blocks of DI-Guy AI Minds.

Sample Code shows implementations of subclasses to perform specific tasks and behaviors. Here is a list of the samples:

class	luaFoodStand	A smart prop that implements a go eat food plan. More...
class	luaLounge	A smart prop that implements a socializing plan. More...
class	luaMiddleEastBaseClass	This base class creates a number of helper functions for the mideastern pedestrian. More...
class	luaMiddleEasternPedestrian	A complex middle eastern pedestrian that can implement a number of behaviors. More...
class	luaMosque	A smart prop that implements a muslim prayer plan. More...
class	luaSimpleFriendlySoldier	
class	luaSimpleOpposingSoldier	
class	luaSimpleSoldier	A simple soldier class to act as sample code. More...
class	luaIED	A class that can help an agent plant an IED, also implements the logic for a vehicle driven IED. used by the middle east pedestrians. More...
class	luaMolotov	A class implements a molotov cocktail that's thrown at something . More...
class	luaPoolOBlood	A class that implements a growing pool of blood typically created by a dieing agent. More...
class	luaProximityMonitor	used by the middle east pedestrians More...
class	luaAngerEvaluator	used by the middle east pedestrians More...
class	luaFocusStack	used by the middle east pedestrians More...

The Middle East Town Demo shows a subset of the sample code classes used by the Middle East Town demo.

Finally, the Handy Utility Functionality has 5 particularly useful classes, lqt for working with our QT User Interface, luaLabel to help with on screen label graphics, luaNavmeshHelper, luaSensorHelper for modifying shaders to simulate sensorized views, and the luaSimModuleHelper for creating ragdoll physics effects for an Agent.

5.1.2 DI-Guy Lua Classes

Following is a list of documented DI-Guy AI lua classes:

<code>coroutine</code>	Lua's built in microthread library
<code>debug</code>	Lua's built in debugging library
<code>diguy_package_registry</code>	A global registry for storing package objects
<code>file</code>	Lua's built in file library
<code>global_meta</code>	
<code>io</code>	Lua's built in input output library
<code>lqt</code>	This package is responsible for loading and wrapping a binding around qt4
<code>lua_character_registry</code>	A global registry for storing <code>luaCharacter</code> objects and descendants
<code>lua_crowd_registry</code>	A global registry for storing <code>luaCrowdObjects</code>
<code>luaAngerEvaluator</code>	Used by the middle east pedestrians
<code>luaCharacter</code>	The base class of all the diguyCharacter's lua representation
<code>luaCrowdObject</code>	A simple object for agents to use as a shared blackboard for storing arbitrary information
<code>luaDirectedAI</code>	
<code>luaFocusStack</code>	Used by the middle east pedestrians
<code>luaFoodStand</code>	A smart prop that implements a go eat food plan
<code>luaIED</code>	A class that can help an agent plant an IED, also implements the logic for a vehicle driven IED. used by the middle east pedestrians
<code>luaLabel</code>	A helper class for setting up camera and other buttons on the screen
<code>luaLifeFormServer</code>	
<code>luaLounge</code>	A smart prop that implements a socializing plan
<code>luaMessageObject</code>	A object that wraps an incoming diguy message
<code>luaMiddleEastBaseClass</code>	This base class creates a number of helper functions for the mideastern pedestrian
<code>luaMiddleEasternPedestrian</code>	A complex middle eastern pedestrian that can implement a number of behaviors
<code>luaMissileRPG</code>	
<code>luaMolotov</code>	A class implements a molotov cocktail that's thrown at something
<code>luaMosque</code>	A smart prop that implements a muslim prayer plan
<code>luaNavmeshHelper</code>	Class that asks the user to create a navigation mesh
<code>luaPoolOfBlood</code>	A class that implements a growing pool of blood typically created by a dieing agent
<code>luaProximityMonitor</code>	Used by the middle east pedestrians
<code>luaScheduleEvents</code>	
<code>luaSensorHelper</code>	This package has functions for setting up shaders for simulated sensor visuals
<code>luaSimModuleHelper</code>	This package includes handy functions for converting characters into ragdolls
<code>luaSimpleFriendlySoldier</code>	
<code>luaSimpleOpposingSoldier</code>	
<code>luaSimpleOpposingSoldierRPG</code>	
<code>luaSimplePedestrian</code>	
<code>luaSimpleSoldier</code>	A simple soldier class to act as sample code
<code>luaSmartObjects</code>	This is a helper class that acts as a registry of objects with plans
<code>luaSmartProp</code>	Base class for creating movable props with auto tagging features
<code>luaStateMachineScript</code>	
<code>luaUIStateMachine</code>	A class for building interactive UI tools
<code>math</code>	Lua's built in math library
<code>os</code>	Lua's built in os library
<code>string</code>	Lua's built in string library
<code>table</code>	Lua's built in table library
<code>type_to_constant</code>	
<code>Vec3</code>	A class for doing basic 3d math in lua

5.1.3 DI-Guy Lua Files

A handy guide to lua files distributed with DI-Guy AI:

c:/bdi/config/diguy/scripts/character_classes/luCharacter.lua
c:/bdi/config/diguy/scripts/character_classes/luDirectedAI.lua
c:/bdi/config/diguy/scripts/character_classes/luFoodStand.lua
c:/bdi/config/diguy/scripts/character_classes/luLounge.lua
c:/bdi/config/diguy/scripts/character_classes/luMiddleEastBaseClass.lua
c:/bdi/config/diguy/scripts/character_classes/luMiddleEasternPedestrian.lua
c:/bdi/config/diguy/scripts/character_classes/luMosque.lua
c:/bdi/config/diguy/scripts/character_classes/luSimpleFriendlySoldier.lua
c:/bdi/config/diguy/scripts/character_classes/luSimpleOpposingSoldier.lua
c:/bdi/config/diguy/scripts/character_classes/luSimpleOpposingSoldierRPG.lua
c:/bdi/config/diguy/scripts/character_classes/luSimplePedestrian.lua
c:/bdi/config/diguy/scripts/character_classes/luSimpleSoldier.lua
c:/bdi/config/diguy/scripts/character_classes/luSmartProp.lua
c:/bdi/config/diguy/scripts/core/additional_docs.lua
c:/bdi/config/diguy/scripts/core/luCore.lua
c:/bdi/config/diguy/scripts/core/luScheduleEvents.lua
c:/bdi/config/diguy/scripts/core/luSmartObjects.lua
c:/bdi/config/diguy/scripts/core/luStateMachineScript.lua
c:/bdi/config/diguy/scripts/core/luUIStateMachine.lua
c:/bdi/config/diguy/scripts/core/vec3.lua
c:/bdi/config/diguy/scripts/utills/lqt.lua
c:/bdi/config/diguy/scripts/utills/luAIED.lua
c:/bdi/config/diguy/scripts/utills/luLabel.lua
c:/bdi/config/diguy/scripts/utills/luLifeFormServer.lua
c:/bdi/config/diguy/scripts/utills/luMolotov.lua
c:/bdi/config/diguy/scripts/utills/luNavmeshHelper.lua
c:/bdi/config/diguy/scripts/utills/luPoolOBlood.lua
c:/bdi/config/diguy/scripts/utills/luSensorHelper.lua
c:/bdi/config/diguy/scripts/utills/luSimModuleHelper.lua
c:/bdi/config/diguy/scripts/utills/mideast_demo_support_classes.lua

5.2 Understanding the Basic Architecture of a DI-Guy AI Mind

There is a great deal of freedom for how one constructs a DI-Guy AI Mind. DI-Guy AI Mind architecture uses a basic paradigm for how minds should work that is simple, powerful, and extensible. It is founded on three key principles:

- 1) STATES – At any given moment, the AI Mind is in one and only one state. Examples of states might include “patrol”, “threaten”, “retreat”, “eat”, “go to church”, or just about any individual task one can think of.
- 2) MESSAGES – Messages are often critical for triggering a transition from one state to another. Messages can come from:
 - a human operator running the simulation
 - other characters in the scene
 - “smart objects” in the scene
 - event callbacks

Examples of messages might be “go patrol” received from an operator, “follow me” from a fellow character, “come pray” from a smart mosque, or a gunshot event in the Agent’s vicinity.

- 3) MESSAGE PROCESSING – Message processing occurs when a message is received. Typically a character is in a state while also listening for messages. When a message is received, it is processed, oftentimes transitioning the character to a new state.

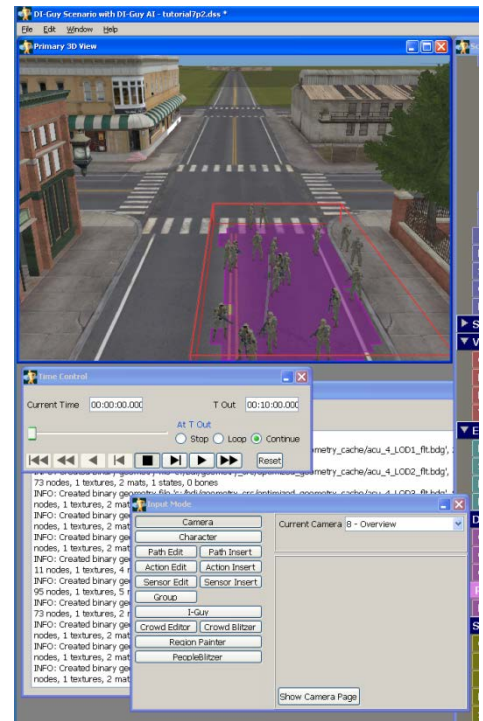
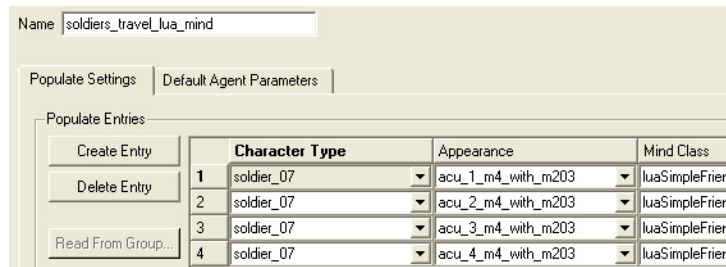
Unlike a factory or a coke machine, the way humans make decisions in the world is very non-linear; DI-Guy AI uses the above architecture because of its strong ability to clearly author minds that reflects this basic reality. It is an approach easy to understand and extensible.

5.3 AI Mind Example – the soldier_travel_lua_mind

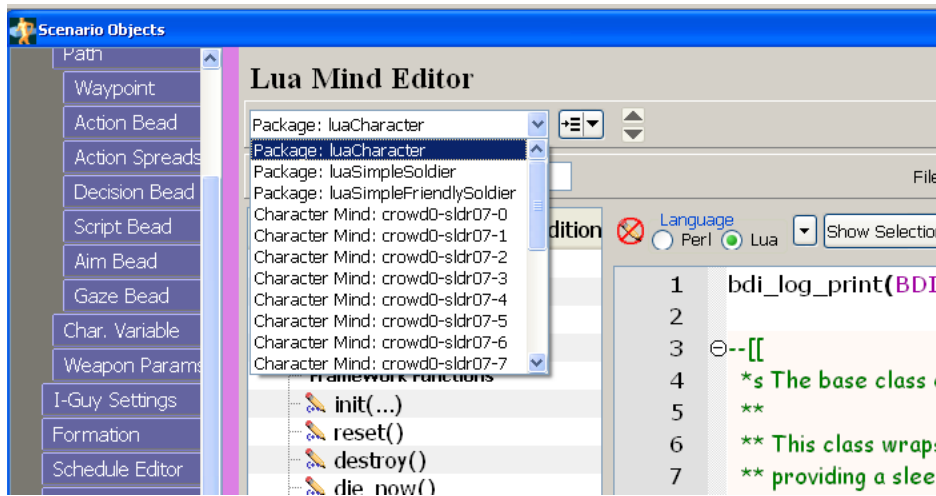
Before we look at a somewhat advanced mind in detail, let’s begin by creating a new scenario and then blitzing in a crowd profile composed of Agents using DI-Guy AI Minds.

1. Start up DI-Guy Scenario.
2. Generate a Global Navmesh called “navmesh” in the Region page of the Scenario Objects window.
3. CrowdBlitz in a set of traveling soldiers using the "soldier_travel_lua_mind" crowd profile.

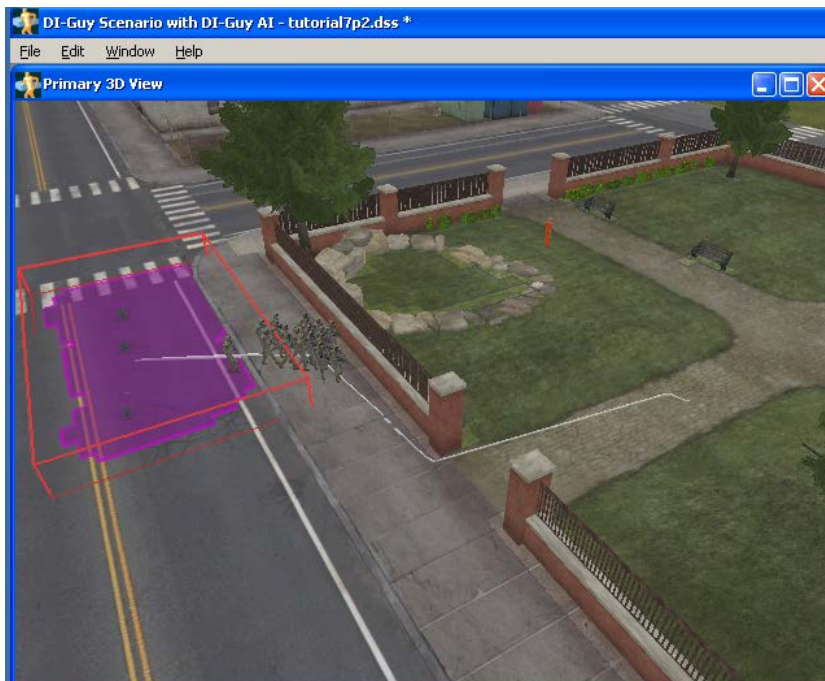
One way to determine which Minds our agents are using is to examine the crowd profile we used “soldier_travel_lua_mind”. The four entries in the Populate Entries table all state that the agents use luaSimpleFriendlySoldier Mind.



Another way is to go to the Lua Mind Editor page of the DI-Guy AI Objects group. If you examine the pull-down list, you will see a set of Character Mind instances, one for each soldier in your crowd.

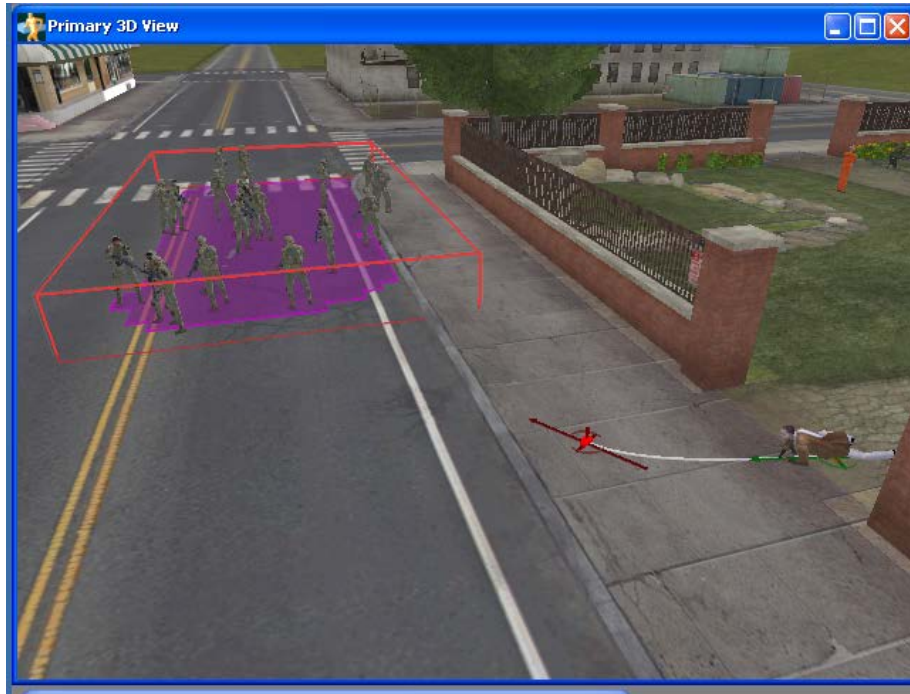


Path Planning – Let’s see the minds in action. Play the scenario. In “Character” mode, right-click on any character in the crowd. Command him to “Move to point” via the popup menu and then select a point inside the navmesh region. You should see the character use path planning to reach his goal.



Reaction to Fire – Let’s have a character fire at the soldiers and see if they react. PeopleBlitz in a soldier_07 character with appearance frank_ak47, making the path a short one that leads towards the

crowd. Then, go to the Character Objects->Character page and select the new character (sldr07-0). Go to the Action Bead page and change the first stand action to be “kneel_aim”. Go to the Decision Bead page and create a decision bead at some time shortly after the character starts kneeling. Tell him to aim at one of the characters in the crowd, and fire his weapon 4 times with a maximum time between shots. Run the scenario. You will see that when the friendly soldier is shot, the other friendlies react and shoot the bad guy.



Frank sneaks up on our crowd.

5.4 The DI-Guy Lua Code Editor

DI-Guy Scenario include a powerful Lua Script Code Editing system that supports code coloring, syntax highlighting, autocomplete and automatic display of DI-Guy function calls. The Code Editor is used for script bead editing and AI Minds. Some of the features of the Editor are described below.

Simple auto complete based on document inspection

Whenever the user begins to enter code in the Code Editor, it will attempt to match other words in the code to the current string being entered. If a potential match(es) is found, the Code Editor brings up a multiple choice box showing those autocomplete words. Note in the image below how sldr07_0 is understood from the earlier context to be a potential match.

```

--sldr07_0
-- this block of code is only called the first time the sldr07_0 is created
⊖ if(sldr07_0 == nil) then
    local character = this_scenario:find_character("sldr07-0");
    sldr07_0 = luaSimpleFriendlySoldier:new(character);

⊖ if(sldr07_0.one_time_init) then
    sldr07_0:one_time_init();
end
-- your one time init code here

sl
sldr07
sldr07_0
sleep
-----
comment
al init arg

```

Autocomplete for DI-Guy Constants

The autocomplete index has also been primed with all of the script accessible DIGUY_* constants. Just type DIGUY to show the list. Callback ID's are also included in the list.

```

7  DIGUY
8  DIGUY_CHARACTER_FEELER_STATUS_NO_CONTACT
9  DIGUY_CHARACTER_FEELER_TURN_DIRECTION_L
10 DIGUY_CHARACTER_FEELER_TURN_DIRECTION_L_OR_R
11 DIGUY_CHARACTER_FEELER_TURN_DIRECTION_NONE
12 DIGUY_CHARACTER_FEELER_TURN_DIRECTION_R
13 DIGUY_CHARACTER_GESTURE_TYPE_COUNT
14 DIGUY_CHARACTER_GESTURE_TYPE_PROGRESSION
15 DIGUY_CHARACTER_GESTURE_TYPE_TABLE
16 DIGUY_CHARACTER_GESTURE_TYPE_UNKNOWN
17
18
19 -- uncomment to show a right click mind menu

```

Autocomplete for DI-Guy object functions

When the member operator “.” is typed in Lua, a simple syntax analyzer is triggered that attempts to derive the object type on the left side of the member operator and then shows what script accessible functions an object has.

- API Based on Known Objects: If an object name is one that is explicitly set by our script engine e.g.: “this_scenario”, “this_character”, “this_app” or “callback_object” the list of functions that that object has should pop up. The callback_object type is derived from the type of callback that is being edited. In this case there is a high level of confidence in the object type that is being picked.
- API Based on Rules: Lua is a dynamically typed language, making it difficult to determine an exact object type except during code execution. We’ve implement a simple rule system for determining what type an object is based on the name of the object.

NOTE: The list below shows what conventions the Code Editor will recognize. Following the code conventions will greatly assist with debugging and rapid prototyping. Let the Code Editor help you. If you have a diguyCharacter, call it “character” or “characterBob”, etc. **USE THE CONVENTIONS!**

Object name starts with:	Assumed to be:
agent or params	diguyAgentParams
camera	diguyViewCamera
camera_settings	diguyViewCameraSettings
character	diguyCharacter
crowd	diguyCrowd
fog	diguyViewFog
gesture	diguyCharacterGesture
group	diguyCharacterGroup
guide	diguyCharacterGuide
impact	diguyImpact
info_popup	diguyInfoPopup
label	diguyViewLabel
light	diguyViewLight
path	diguyCharacterPath
profile	diguyCrowdProfile
sensor_region	diguySensorRegion
region	diguyRegion
signal	diguySignal
sound	diguySound
view	diguyView
variable	diguyVariable
waypoint	diguyWaypoint

These conventions match code automatically generated by our decision bead to script bead converter.

```

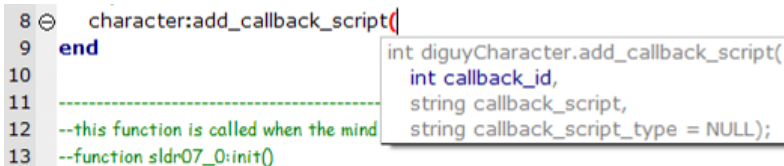
8   character:
9   end
10
11  -----
12  --this func add_all_gestures
13  --function add_callback_script
14  -- call add_guide
15  -- luaS add_motion_texture
16  add_pending_desired_action
17  add_pending_force_action
18  add_secondary_heartbeat
19  add_state_machine_instance
20  add_wakeup_callback
21  agent_accept_message

```

Example of diguyCharacter object autocomplete list

Calltips

Whenever “(” is entered, the code editor will show calltips for the function in question:

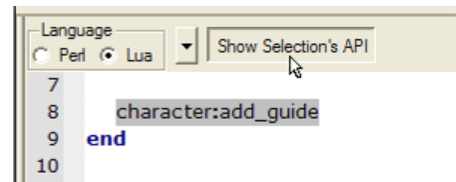


The calltip information is found in (\$DIGUY)\doc\diguy_sdk_reference\api. The ESC key can be used at any time to hide the calltip.

Full Documentation in the API Viewer

The Code Editor includes a simple DI-Guy API documentation viewer. The documentation is accessible directly via Help->API Docs or indirectly via the **Show Selection's API** button.

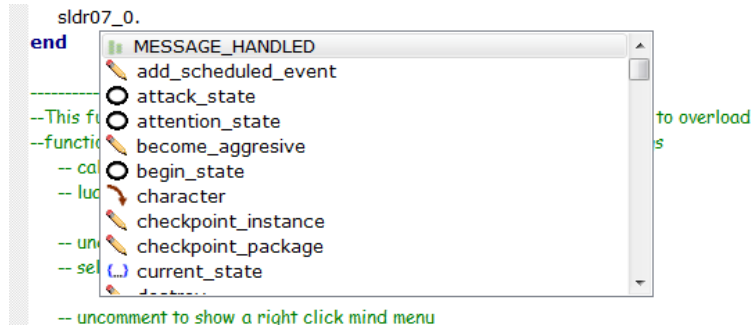
EXAMPLE: If *character:add_guide* is selected and then **Show Selection's API** is clicked, DI-Guy Scenario will launch the DI-Guy API viewer and scroll to the object/function entry. Note this will only work with objects that match the naming rules above.



Querying Lua for related objects and fields

The Lua language has introspection functionality, allowing users to query what objects and fields a given object might have. If it is not possible to guess an object's type, the Lua interpreter is queried for an object's existence.

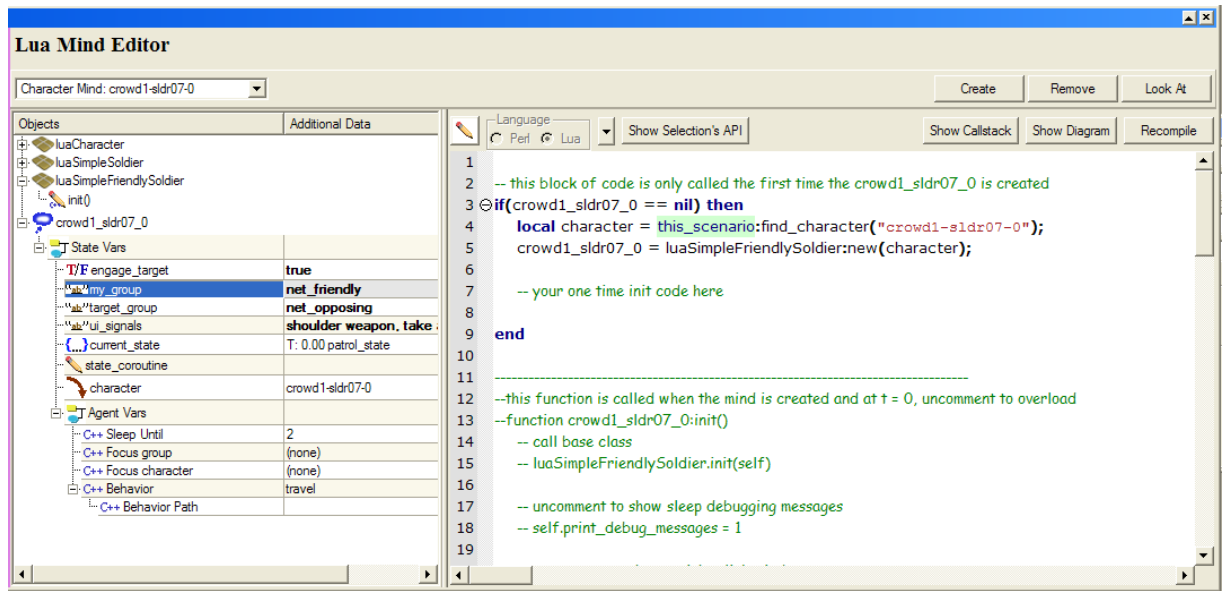
EXAMPLE: If “*luaCharacter:*” is typed a function (located in luacore.lua) recursively iterates over the luaCharacter object's functions and adds them to a list which is provided to autocomplete. A similar thing happens if “*luaCharacter.*” is typed in, the autocomplete list will include all fields including strings, booleans, numbers, tables, as well as the functions that the object has.



The parser will attempt to identify nested luaObjects as well: e.g. `luaSmartObjects.registry.socialize_men`

Nested luaObject autocomplete also work on any built-in lua library. Therefore, if the user types “`math.`”, “`string.`”, or “`table.`”, the editor will bring up these built-in library contents. Calltips for built-in lua library should appear as well. Note that unfortunately a local variable cannot be analyzed in this fashion.

5.5 Lua Packages



The Lua Mind Editor Interface

In DI-Guy AI, a package is a Lua script, either on disk (in the \$DIGUY/config/diguy/scripts/<subdirectory> folders) or coded directly in a scenario file script, that is automatically compiled at scenario load or whenever the script changes, and supports a notion of

dependencies. Typically, packages contain code that is run at initialization as well as code that executes at runtime.

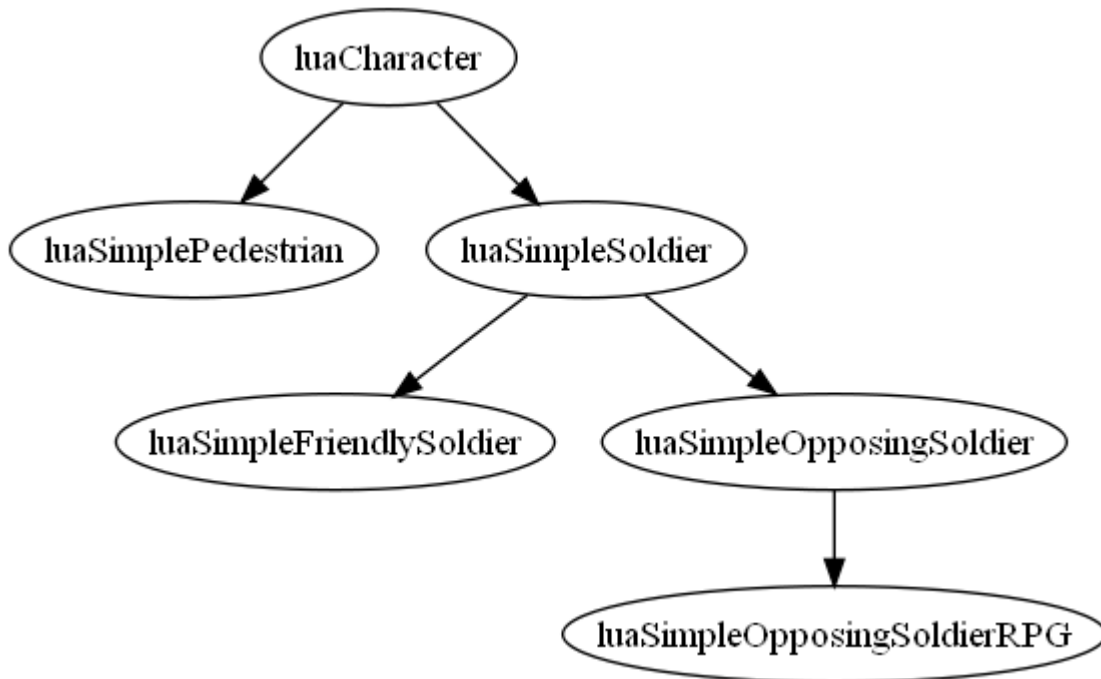
Note that changes to the on-disk packages can be seen and used by other scenarios, while the in-scenario packages are local to the specific scenario. The Convert button (Convert To Local Script/Convert To External Script) allows packages to be imported for editing and exported for sharing.

In the Lua Mind Editor page, we can see from the pull-down that a total of 4 packages were loaded when we blitzed in the soldiers:

- luaCharacter
- luaSimpleSoldier
- luaSimpleFriendlySoldier
- luaNavMeshHelper

Each package is a single Lua script file containing multiple functions that represent a particular Mind.

Let's ignore the luaNavMeshHelper package and instead look at the first three packages. We will see that luaCharacter serves as a base class for DI-Guy AI Minds. luaSimpleSoldier is a subclass of luaCharacter. Meanwhile, luaSimpleFriendlySoldier is a subclass of luaSimpleSoldier. Following is a diagram of class inheritance for the AI Minds offered in DI-Guy AI.



The following is a cursory overview of the luaCharacter, luaSimpleSoldier, and luaSimpleFriendlySoldier packages. We are going to first try to understand the basics of what it going on, without delving into all the details.

The luaSimpleFriendlySoldier Package

Using the scenario you created in the previous tutorial, select luaSimpleFriendlySoldier from the top pull-down in the Lua mind editor.

LuaSimpleFriendlySoldier is a subclass which inherits from the luaSimpleSoldier class.

```
luaSimpleFriendlySoldier = inheritsFrom( luaSimpleSoldier,
"luaSimpleFriendlySoldier" )
```

The luaSimpleFriendlySoldier package contains a single generic function: init(). A generic function is indicated by the pencil icon in the left side browser. You'll notice that there is also a small "c++" on the pencil icon; this indicates that the DI-Guy Mind framework will automatically call this Lua function. In other words, init() is one of a small number of reserved function names that mean something special to DI-Guy AI. Users can also define generic functions that are not framework functions. These will not include the C++ marking.

A quick perusal of the init function shows that the luaSimpleFriendlySoldier class adds the ability to identify itself as a friendly force and identify opposing forces as the enemy. As you can imagine, luaSimpleOpposingSoldier uses the opposite assignments.

The luaSimpleSoldier Package – first look

Now select luaSimpleFriendlySoldier from the top pull-down.

The luaSimpleSoldier is more complex. This is where all the basic soldier Mind functionality is specified.

The luaSimpleSoldier contains two framework generic functions, init() and init_package(), plus two other generic functions, move_to_background_check() and become_aggressive().

Next you can see that two callback functions are identified for when a target is acquired and when an impact occurs. The icon for a callback function looks like a telephone in the left side browser. The browser identifies these functions as callbacks because they use the convention on_<text> for the function name. Note that labeling a function as a callback function by using the on_<text> convention is somewhat of a style choice. The first callback, on_impact_callback(), is a classic DI-Guy event and callback. The second callback, on_target_acquired(), is instead an event invented by the author of this class, when he determines that the event has occurred, he calls on_target_acquired().

The Lua Mind Editor has a lot of nice browser features that help make your AI Mind understandable such as the different graphic icons for different types of functions, but these only work *if you follow the recommended conventions*. Doing so will make your work as an advanced AI Mind programmer much simpler. Later in this document, we will provide a thorough list of recommended conventions.

The next set of functions is the message handling functions. They use the convention process_messages_<text> for their Lua function names, and are represented iconically with the  "branching" icon.

The entry_point function indicates where the State Machine for the character begins. It is represented by the traffic light “Go!” icon.



Finally a large number of state functions are listed. State functions self-identify themselves by the use of the string “state” in their function names and are represented with the circle icon.



You can display a state diagram of the luaSimpleFriendlySoldier’s State Machine by pushing the “show diagram” button. (NOTE: you need to first install the freeware package graphviz which we include in our 3rdparty directory; make sure its bin directory is in your path before launching DI-Guy Scenario.) For a state diagram to function properly don’t forget there should be an entry point function as a starting point.

The luaCharacter Package

LuaCharacter is the base class for all DI-Guy AI Minds. *You should not need to edit this class* (in general you should make new subclasses). But it is useful to browse the class to see the base functionality it enables, how it sets up your subclasses for overrides, and automatically gets hooked up to the DI-Guy simulation engine:

reset()	Called every time you hit reset in DI-Guy Scenario or call reset() in the SDK.
init()	Called at time 0 and when the mind is created.
destroy()	Called when the character is destroyed.
die_now()	Called when the character is about to be killed, this is an appropriate place to notify fellow group members, hide labels, etc.
sleep() sleep_ignore_messages() sleep_process_messages() sleep_until() wait_for_message()	A number of sleep and message-related functions. One of the powers of Lua in DI-Guy is its ability to efficiently sleep and boost performance. In addition, DI-Guy AI minds offer a sophisticated messaging system to aid in inter-entity communications.
entry_point()	Starting point for the character’s state machine.
state_manager()	Support for the character’s state machine.
draw()	A function called during the scenario rendering; when overridden, allows custom graphics to be drawn in the 3D view.
get_user_selected_point() get_user_selected_character()	Helper functions when working through the DI-Guy Scenario GUI.
simple_move_to_state()	All AI Mind Characters have the “Move to” state built-in.
begin_state()	A helper function that sets the character’s current_state field; useful for processing messages and debugging. Recommended first line in all state functions.

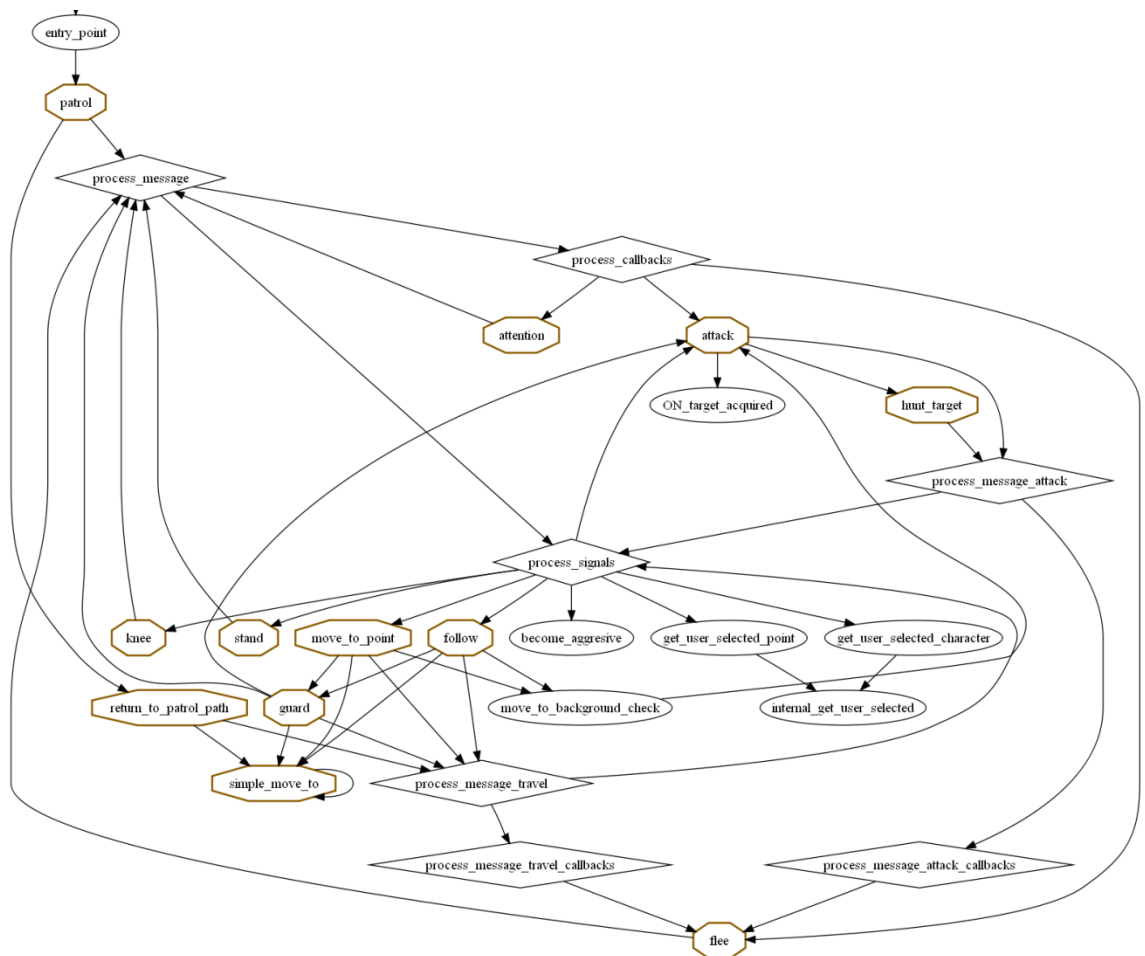
5.6 Visualizing the State Machine

The “Show Diagram” button on the Lua Mind Editor page lets you visualize a package’s State Machine. Here is a guide to understanding what the different shapes in the diagram mean:

GOLD OCTOGONS – gold octagons indicate DI-Guy AI Mind States. Note that the “_state” at the end of the functions are trimmed off for brevity’s sake.

BLACK DIAMONDS – indicate DI-Guy AI Mind Message Processing Functions.

ARROWS – indicate that the target function is called from the source function. Note that this does not preclude returning from that function back to the source. For example, in the soldier diagram below, return_patrol can lead directly to patrol. When you select more complex diagrams, you will also see individual Messages and Callbacks added into the state machine.



Above is the State Machine Diagram for luaSimpleSoldier. Here are some interesting points to note:

1. The State Machine always starts at the Entry Point state.

2. The first state is patrol.
3. The default process_messages function calls the default process_callbacks function and default process_signals function. The specialized process_attack_messages calls process_attack_callbacks but then just uses the default process_signals function. The specialized process_travel_messages is similar to attack, it calls a specialized process_travel_callbacks but then just uses the default process_signals.
4. The topology of the State Machine Diagram should be ignored. That is, there is no particular significance that the flee state is at the bottom of the diagram while the attention state is near the top.
5. Attack and Hunt target use the specialized process_messages_attack message handlers.
6. Move_to_point, follow, guard, and return_to_patrol_path use the specialized process_messages_travel message handlers.
7. Generic functions such as move_to_background_check are shown with circles.

5.7 luaSimpleSoldier Package: A Second Look

Let's look more closely now at the luaSimpleSoldier package functions, and see if we can test some of the functionality:

init_package() – This code checks if a navigation mesh is associated with the character. It is what invoked the fancy navigation mesh initialization when you first blitzed in the crowd.

init() – The Impact, Crowd Member Killed, and Nearby Weapon Fired events are mapped as wakeup callbacks for luaSimpleSoldier. This means that the active process_message function will be called when these events occur.

We also want the Impact event to call our on_impact_callback() function. We have set this up with a little indirection because the impact callback returns a diguyCharacter object, and we need to find the Lua character that corresponds to it. (Note that a luaCharacter is a wrapper class that contains a pointer to the underlying diguyCharacter class along with other useful information). It uses the Lua core function find_lua_character() to do this. Another purpose of this callback script is to override the luaCharacter impact callback script, which automatically kills the soldier. Our callback is a little more complex than that.

Now let's look at the entry_point function. This is the function that is called when the character starts up as the beginning place for its State Machine.

State Functions – entry_point

```
function luaSimpleSoldier:entry_point()  
    self:patrol_state();  
end
```

The entry_point function is very simple. It immediately puts the character into patrol_state.

State Functions – patrol state

Let's examine the State Function: patrol_state.

```
function luaSimpleSoldier:patrol_state()
  --[[
  it's worth pointing out that by default the soldier will do what the
  Initial crowd UI specifies.
  ie wander or mingle or travel. It's only once events and messages start
  that things change.
  the return_to_patrol_path_state attempts to put the character back to
  it's initial state when the message was received
  ]]

  -- update the current state (useful for debugging)
  local state_info = self:begin_state("patrol_state");

  -- let the gui know what signals we handle
  self.ui_signals = "shoulder weapon, take a knee, move to point, move to
path";
  -- This is the default state of the soldiers
  while 1 do
    -- wait for a signal. when we get it pass it to process_message
    local ret_val = self:process_message(self:sleep(2000));

    if(self.print_debug_messages == 1 and ret_val) then
      print(self, "res is", ret_val);
    end

    if(self.is_new_state(ret_val) == true) then
      if(self.print_debug_messages == 1 ) then
        print(self, "Starting ", self:get_state_name(ret_val.state),
          "caused by", self.last_message) ;
      end
      -- transition to the new state
      return self:invoke_state(ret_val);
    end

    if(ret_val == "exit" or ret_val == "restart" or ret_val == -1) then
      self:return_to_patrol_path_state();
      return self:patrol_state();
    end
  end
end
```

State functions typically have 3 areas:

- 1) Begin Statement – The self:begin_state("<name of state>"). While not necessary, it is strongly recommended that all state functions have this opening line. It is useful for debugging purposes.

- 2) Parameterization, behavior, and logic section. Often the bulk of the state, this is where parameters are altered, DI-Guy SDK functions are called, and logic is performed.
- 3) Process messages handoff. Once you have set up your behavior, you want to set into action the process_messages message handlers that will process any interrupt activity during the time the character is in the state.

In the patrol state function, we can see that we first set the begin state to “patrol state”. We then set some parameters governing the on-screen GUI pull-down. Finally, we call process_messages after sleeping every two seconds. Note that when some particular returns from process_message are passed back to patrol_state, the character will invoke the return_to_patrol_path state, and upon return from that, revert back to patrol state.

Sometimes states will progress to another state without a process messages call (when they are doing something simple) or upon return from a process messages call (when they know there will be a next logical state upon return).

The Default Message processing function – process_messages

The process_messages function serves as the default message processing function for the luaSimpleSoldier. It calls the following two subordinate functions for handling the various types of messages that may be called.

The Default Callback Message processing function – process_messages_callback

This function processes any callback events triggered.

```
function luaSimpleSoldier:process_callbacks(message_object)
    --print(self, self.current_state.name, " got message ", message_object);

    if(self.current_state.name == "flee_state") then
        return luaCharacter.MESSAGE_HANDLED;
    end

    local sender          = message_object["sender"];
    local message_type    = message_object["message_type"];
    local message         = message_object["message"];
    local message_params  = message_object["params"];

    -- THIS IS THE DEFAULT CALLBACK HANDLER
    if(message == "CALLBACK_ID_AGENT_NEARBY_WEAPON_FIRED" or
       message == "CALLBACK_ID_AGENT_NEARBY_SCENE_OBJECT_IMPACT" or
```

```
message == "CALLBACK_ID_AGENT_CROWD_MEMBER_KILLED" or
message == "CALLBACK_ID_AGENT_NEARBY_WEAPON_FIRED") then

    local crowd = self.character:agent_get_crowd();
    local impact = crowd:get_callback_impact();
    local character_attacker = impact:get_attacker();
    -- if more than 80% of the crowd is dead, flee
    if(crowd:is_crowd_dead(.80) == 1) then
        return self:flee_state();
    end
    if(character_attacker:is_group_member(self.target_group) == 1 and
        self.engage_target == true) then
        -- the shooter is a member of the group that is my enemy I
        -- should probably just attack
        -- pushdown attack
        return self:attack_state(character_attacker,
            character_attacker,
            "group");
    else

        if (message == "CALLBACK_ID_AGENT_CROWD_MEMBER_KILLED" or
            self.current_state.name == "attention_state") then
            -- ok check if it's friendly fire
            if(character_attacker:is_group_member(self.my_group) == 0) then
                -- nope... attack
                if(self.current_state.name == "attention_state") then
                    -- we're already in a push down cause it to pop up
                    return {state = self.attack_state, arg1 = character_attacker};
                end
                return self:attack_state(character_attacker);
            end
        else
            local dist =
```

```

        self.character:get_distance_to_character(character_attacker);
        -- just a fired weapon, might be harmless
        if(math.random() > dist/25 ) then
            --another push down state
            return self:attention_state(character_attacker);
        end
    end
end

else
    local str = tostring(self) .." no handler for callback " ..
        message .."\n".. debug.traceback();
    this_scenario:lua_log_printf(BDI_LOG_WARN, str );
    print(str);
end
end
end

```

There are some interesting things to observe in this function:

1. The process_callbacks function shows how to unpack a message_object to get at important constituent parts such as “who sent the message”, “what type of message was it”, “what is the message”, and any parameters associated with the message.
2. The function only handles four types of messages. Anything else and it prints an error message.
3. A call to the DI-Guy SDK is made to determine what crowd the character belongs to.
4. An impact descriptor from the shot fired is retrieved. From that, the attacker is determined.
5. If his crowd is more than eighty percent dead, the soldier flees.
6. If the shot was due to enemy fire, the soldier is switched to attack state.
7. If someone was killed and the attacker was a neutral, the soldier is switched to attack state.
8. If it was just a nearby shot, switch to attention_state to look at the attacker if the shot is close by.

The Default Signal Message processing function – process_messages_signals

This function processes any messages received either as a signal from the UI or as message broadcast from another entity in the scene.

```

function luaSimpleSoldier:process_signals(message_object)
    --print(self, "got message ", message_object);

    local sender          = message_object["sender"];
    local message_type    = message_object["message_type"];
    local message         = message_object["message"];
    local message_params  = message_object["params"];

    if(message == "shoulder weapon") then
        -- stay in this state but modify our behavior to become aggressive
        self:become_aggressive();
        return
    elseif(message == "take a knee") then

        -- Return a table saying which state the calling state should
        -- transition to next. This will cause us to leave the calling
        -- state
        return {state = self.knee_state}

        ...

    elseif(message == "shoot someone") then
        -- we're going to transition to a the attack_state temporarily without
        -- leaving the calling state. you need to be very careful doing this
        -- type of state transition as you can end up with an infinite
        -- recursion (and a stack overflow) if the new state can get back
        -- to the old state.
        -- this type of transition is useful however when you want to come
        -- back to the calling state without resetting its internal
        -- parameters
        -- push down
        local target = self:get_user_selected_character();
        return self:attack_state(target, "just_one");

    elseif(message == "attack") then
        -- you can pass parameters to the state that is to be invoked
        return {state = self.attack_state, arg1 =
            this_scenario:find_character(message_params)}

        ...

    elseif(message == "stop attacking") then
        -- here we return a token to the calling state telling it to exit
        self.engage_target = false
        return "exit"

    elseif(message == "move to point") then
        local vis = this_scenario:get_lua_objects_visible();
        this_scenario:set_lua_objects_visible(DIGUY_VISIBLE_ALL);
        self.draw_ui = self.draw_point_select;
        local x,y,z = self:get_user_selected_point();
        self.draw_ui = nil;

```

```

    this_scenario:set_lua_objects_visible(vis);

    return {state = self.move_to_point_state, arg1 = Vec3:new(x,y,z),
           arg2 = message_type}

...
elseif(message == "destroy squad") then
    -- here we use an api call to edit the scenario
    this_scenario:destroy_crowd(self.character:agent_get_crowd(), 1 ,1);
else
    print(self, "no handler for signal '", message, "' from: ", sender);
end
end
end

```

Some things of note:

1. The code handling for many signals has been elided here as a majority of signals are processed in the same way.
2. Some signals require additional information. For example, move to point invokes the `get_user_selected_point()`, a GUI process meant to run in DI-Guy Scenario. Thus, in this case, the mind actually prompts the user for information.
3. Return to patrol, when gotten as a signal from the GUI, broadcasts a message to fellow agents within 10 meters of the character to also return to patrol.
4. Calling a state directly should only be done if you know that the called state will not call this state through some loop in the finite state machine. This will cause the Lua stack to grow indefinitely.
5. Returning a table containing a state function to call will cause the calling state to fully exit and invoke the new state function. This clears the calling state off of the Lua stack.

Non-default Message processing – process_messages_attack

Sometimes, a particular state will not want to process new messages using the default message processor. When that is the case, you can setup completely different message processing functions, and then have the calling state activate that message processor. The `attack_state` is a good example of this; it calls `process_messages_attack` instead of `process_messages`. That function calls a different callback message handler, and then uses the default signal handler. Travel is similar.

Patrol – This is the default State. The state is entered immediately from the Entry Point. Calls into `process_messages` will send the soldier to different states. He can then eventually return to Patrol State, sometimes directly, and sometimes via Return to Patrol State. During Patrol State, the soldier uses the incoming base behavior.

Follow – The soldier doesn't actually follow the sender! Instead, he queries where the sender is going, and then goes there via the `simple_move_to_state()` function. When he arrives, he enters Guard State

Guard – A soldier entering guard state sets his desired position to be his current position. He calls `process_messages`, waiting for new interrupts. He also checks if there are any enemy targets. If there are, he enters an attack state, and upon return from the attack state, uses `simple_move_to_state` to return to his guard position.

Return to Patrol – During execution of a state, the character may have moved away from his original patrol area. This state executes a `simple_move_to_state()` to return the character to a position near the patrol area.

Move to Point – The soldier adds “return to patrol” to his UI list, queries the operator for a move-to point, signals other members of his group to follow him, and then performs a `simple_move_to_state()`. Upon completion, he enters Guard State, i.e. he remains at the point he was sent to.

Hunt Target – The soldier uses the navmesh to navigate to where he can see the target.

Attack – Based on incoming arguments, the soldier determines a target character. He adds a wakeup callback function for `AGENT_OUT_OF_TARGETS` in case there are no available target characters. He sets up use of his specialized message processing function, `process_message_attack()`. He then begins a while loop where he periodically checks on the status of his target. If the target is dead, he will find another target if he has a designated target group, or else return. If the target is alive and he can’t see him, he enters the Hunt Target State. If the target is alive and he can see him, he activates his attack behavior and calls the `ON_target_acquired` function. By default this just puts the character into the low level attack behavior; however the RPG descendant class overrides this function and fires a missile when `ON_target_acquired` occurs.

Stand – Resets the UI, sets posture and variants to upright and normal, and broadcasts some “angry” and “look at me” signals to those nearby.

Knee – Resets the UI, sets posture and variants to crouched and normal, and broadcasts some “happy” and “look at me” signals to those nearby.

Attention – The soldier will look at the attention target.

Flee – The soldier sets the target character as his focus and then flees 7 meters from him. He returns an “exit value”.

5.8 Understanding how DI-Guy AI Minds Sleep

One of the essential aspects of proper DI-Guy AI Mind design is their need to act as an extremely efficient high-level controller; after all, your world may be filled with hundreds of these characters, all thinking simultaneously. Therefore, DI-Guy AI Minds need to be extremely efficient and only be active and expend CPU time when decisions need to be made. The rest of the time, the Mind should go to sleep.

When a Mind goes to sleep 1) a timer is set indicating when the character is to be woken up and 2) the character’s co-routine or microthread is suspended. The second bit is where the performance benefits occur. When the timer times out, the microthread is resumed and the mind script continues where it left off. This is an essential aspect of DI-Guy AI Minds; it allows complex time code to be carried out in a simple linear script.

The sleep cycle can also be interrupted before timing out by an incoming message. There are a number of different ways such a message may be generated:

- Signals from the user interface (e.g. -Right Click on character GUI)
- Messages sent by another character or smart object via functions such as `agent_broadcast_message()`, `agent_broadcast_message_to_group()`, or `agent_accept_message()`.
- Messages triggered by callbacks; when a script calls `diguyCharacter::add_wakeup_callback()` it effectively requests that whenever the specified callback is triggered, the character is woken up with a message.

When a message is received it is converted into a message object and returned by the sleep function. If the sleep function times out without receiving a message, it will not return a message object. The following code snippet shows a mind that sleeps for 30 second intervals. No message object will be returned if the sleep was not interrupted. If it was interrupted, the `message_obj` exists and `process_messages` is called.

```
while (1) do
  local message_obj = self:sleep(30);
  if(message_obj) then
    self:process_messages(message_obj)
  end
end
```

Here is a creative example of using sleep to efficiently control the RPG character:

```
function luaRPG:wait_for_target_state()
  while (1) do
    self.current_state = "waiting for target"
    self.ui_signals = "stop";

    if(self:process_messages(self:sleep(2)) == "exit") then return end;

    local character_target =
      self.character:get_nearest_active_character_in_group("net_friendly",1);

    if(character_target) then
      self:fire_rpg_state(character_target);

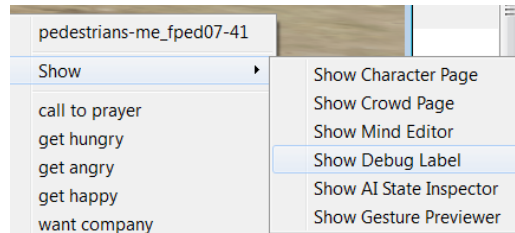
      if(self:sleep_process_messages(20, self.process_messages_attack)) then
        return end
      end
    end
  end
end
```

In this example, the “waiting for target” state is entered. The character sleeps for two seconds, only being woken up by incoming messages. When it does return, either because 2 seconds have elapsed or an incoming message interrupted the sleep, the RPG character checks the return status. If it is “exit”, he returns out of the “waiting for target” state. If not, he determines who is the closest nearest character belonging to the “net_friendly” group. Note that the second argument, 1, indicates that there must be a Line-of-Sight (LOS) to the character. If a target is acquired, the soldier enters his `fire_rpg_state`. Upon

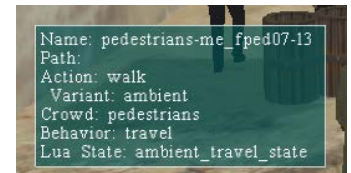
return, the RPG soldier waits 20 seconds before again looking for a target, to simulate his need to reload, etc.

5.9 Debugging Minds

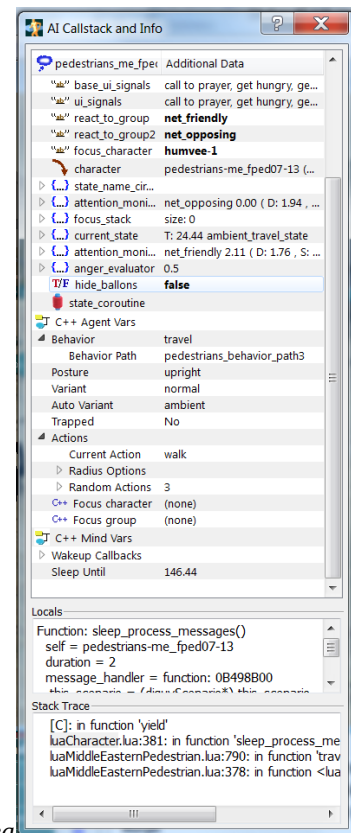
DI-Guy has many ways to debug minds at any point you can right click on the character:



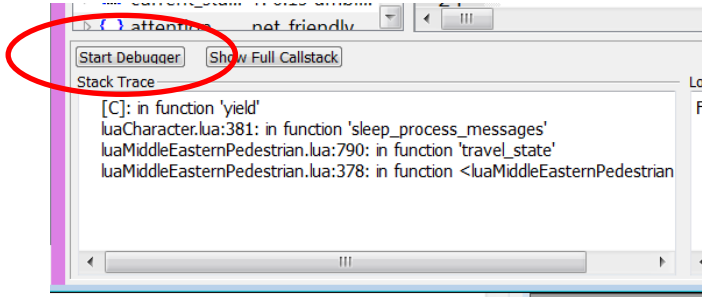
- **Show Debug Label** will turn labels on for the character and show a simple list of things like the character's Path, Action, Behavior, etc, the function `luaCharacter:get_ui_state_label()` can also be overloaded and its return value will get added to the debug label.



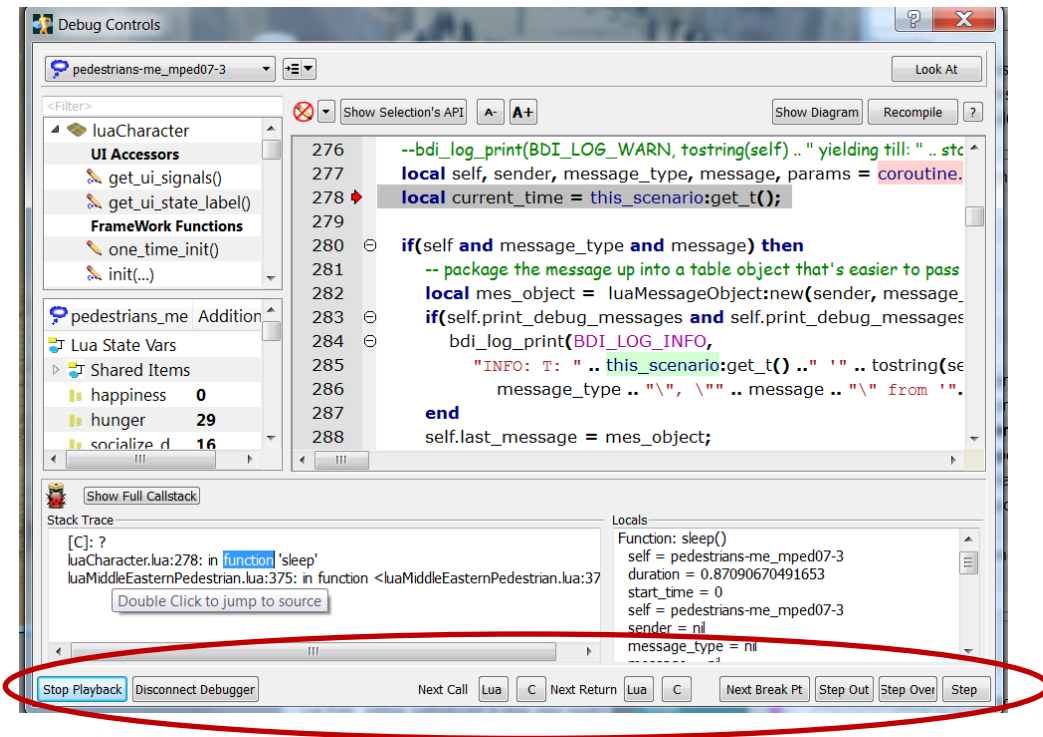
- **Show AI State Inspector** the AI state inspector is a streamlined view of the current state of the character's base behavior C++ state as well as lua state in a compact user interface. It should update every 2 seconds with whomever was clicked last.



- **Interactive Debugger** Trying to understand the flow of messages for an AI, select the AI in question, right click, do show mind editor and click start debugger:



This will pop up an interactive step-by-step debugger that can be used to examine messages and the logic that is triggered when they come in:



5.10 Mind References

This section provides detail on some miscellaneous features of DI-Guy AI Minds. Where appropriate, information on accessing these features from the SDK is also given.

luaCharacter.init_code – the constructor template string. When character inherits from the luaCharacter base class, this string is used as a template for creating the mind script. Users can overload this in a subclassed package as desired.

self.ui_signals – this is a comma delimited list of signals that the agent can currently accept. These signals are reflected in the right-click pulldown menu when in character mode in the DI-Guy Scenario 3-D Window. Typically it is updated when a character enters a new state. While by default this is a string, overloading luaCharacter:get_ui_signals() allows the end user to build a more complex representation of a signal list, as long as the function returns a string.

To duplicate this behavior in the SDK call:

```
diguyCharacter::evaluate_mind_function("get_ui_signals", NULL, 1);
```

Currently a ‘-’ indicates that a separator should be added to the menu.

self.ui_state_label – this simple string will be appended to the state text label when a character has the **Name Label Shows Character State** option selected. While by default this is a string, overloading luaCharacter:get_ui_state_label() allows the end user to build a more elaborate string.

To duplicate this behavior in the SDK call:

```
diguyCharacter::evaluate_mind_function("get_ui_state_label", NULL, 1);
```

Objects with a tostring function.

When an object has a table as a field you can give a hint to the Lua Mind editor on how that data should be visualized by adding a tostring function to the object:

```
function luaCharacter:begin_state(_name)
    if(not self.current_state) then
        self.current_state = { name = _name, t = this_scenario:get_t()};
        self.current_state.tostring =
            function (e)
                return "T: " ..string.format("%3.2f",e.t) .. " " .. e.name;
            end
        end
    end
end
```

The above code snippet shows how current state is updated with a more complex string.

ui_signals	shoulder weapon, take a knee, retreat, m...
current_state	T: 0.00 patrol_state
state_coroutine	

6. DI-Guy AI FAQ

Following are some frequently asked questions about DI-Guy AI:

1. What happens to float * and int * between C++ and Lua

Float * and int * are converted to lua return values so

```
int diguyCharacter get_position(float* tx, float* ty, float* tz);
```

is actually called in lua as

```
local res, x, y, z = character:get_position();
```

2. Why are my if statements always true

When a C++ function returns any value lua consider the following to be true

```
if(character:is_group_member("pedestrians"))
```

- ALWAYS True because it returned something

```
if(character:is_group_member("pedestrians") == 1)
```

- True when you expect it to be true

3. ***I can see my agents but they are not moving.*** Check the path shape the crowd behavior is using if your Agent's behavior is Travel. Sometimes a zero length path shape can be generated or associated with the path by accident.
4. ***I am using the attack behavior but my agents don't attack.*** Make sure that your still action is an aim action such as kneel_aim, stand_aim, etc, or that you have selected <default> for your still action and have selected the "aim" variant. Automatic variation picking should fix this problem in DI-Guy 12.5.

