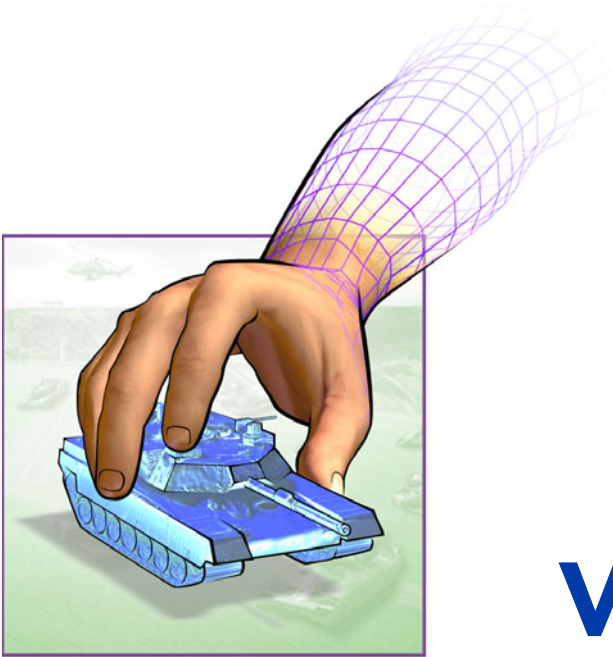
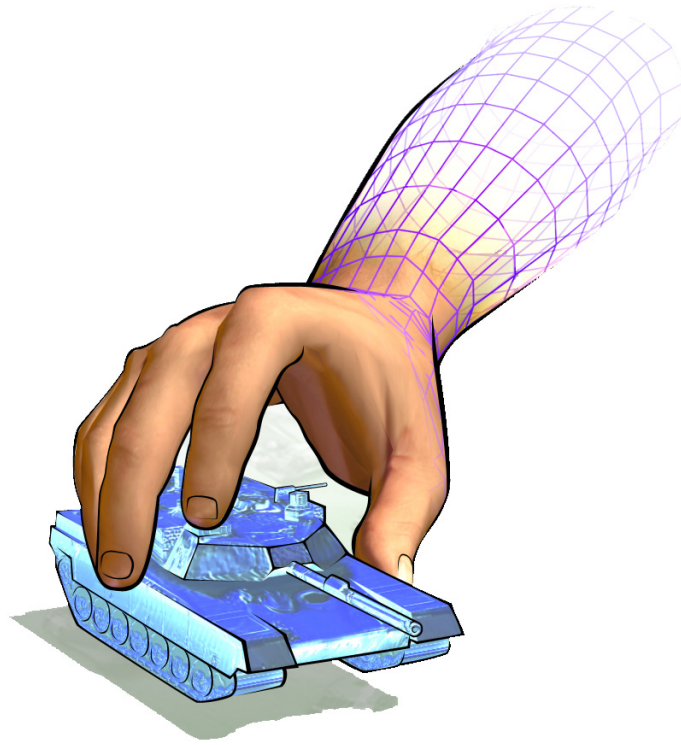




VR-Forces

Migration Guide



VR-Forces

Migration Guide

Copyright © 2009 VT MÄK
All rights Reserved. Printed in the United States.

Under copyright laws, no part of this document may be copied or reproduced in any form without prior written consent of VT MÄK, Inc.

VR-Exchange™ and VR-Vantage™ are trademarks of VT MÄK.
MÄK Technologies®, VR-Forces®, RTIspy®, B-HAVE®, and VR-Link® are registered trademarks of VT MÄK.

GL Studio® is a registered trademark of The DiSTI® Corporation.

Portions of this software utilize SpeedTree®RT technology (©2008 Interactive Data Visualization, Inc.). SpeedTree® is a registered trademark of Interactive Data Visualization, Inc. All rights reserved.

SilverLining™ is a trademark of Sundog Software.

All other trademarks are owned by their respective companies.

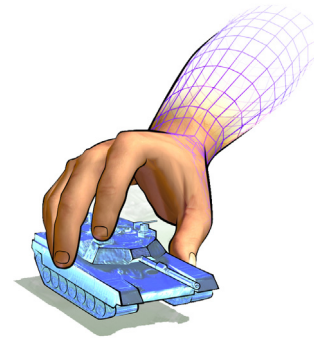
For third-party license information, please see [“Third Party Licenses,”](#) on page 1-2.

VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

Voice: 617-876-8085
Fax: 617-876-9208

info@mak.com
www.mak.com

Revision VRF-3.12-9-090601



Contents

Preface

VR-Forces Documentation	12
Online Manuals.....	12
MÄK Products	13
How to Contact Us	15
Document Conventions	16
Hardware and Operating System Naming Conventions.....	16
Mouse Button Naming Conventions	17
Third Party Licenses	17
Boost License.....	17
libXML and libICONV.....	18
pThreads Library	18
EHS, PCRE, and PME.....	18

Chapter 1 Migrating from 2.x to 3.0

1.1. Updated Simulation API	1-2
1.2. Changes to the GUI/Simulation Engine Communications	1-3
1.3. Improved Performance	1-3
1.4. Miscellaneous Toolkit Changes	1-4
1.5. Remote Control API	1-4

Chapter 2 Migrating from 3.0 to 3.1

2.1. The VR-Forces GUI API	2-2
2.2. Overview of the Main Classes in the GUI API	2-2
2.2.1. The Top-Level Classes	2-2
2.2.2. Other Important Classes	2-3
2.2.3. Customizing or Extending the VR-Force GUI Application	2-4
2.2.4. Miscellaneous Qt Issues	2-5
2.3. Building the vrfGui Application and Examples	2-7

2.4. Introduction to the GUI API Examples	2-7
2.4.1. The Fog of War Example	2-7
2.4.2. The Add Task Example	2-9
2.4.3. The Simple Scenario Player Example	2-13
2.4.4. The Draw Missile Impact Example	2-14

Chapter 3 Migrating from 3.2 to 3.3

3.1. Changes to Entity and Object Naming	3-2
3.1.1. Backwards Compatibility of Object Names	3-2
3.2. Backward Compatibility of Object Parameters Database Files	3-4
3.3. Simulation API Changes	3-5
3.3.1. Changes to Weapons Systems	3-7
3.3.2. Other Changes Related to Weapons Systems	3-8
3.3.3. Changes to the Organization Manager	3-8
3.3.4. Changes to Radios	3-9
3.3.5. Miscellaneous Simulation API Changes	3-10
3.4. Changes to the Remote Control API	3-10
3.5. Changes to the GUI API	3-10
3.6. Porting MÄK Plan View Display Examples to VR-Forces	3-11

Chapter 4 Migrating from 3.3 to 3.4

4.1. Updated GUI API	4-2
4.2. New Formats for Scenario Files	4-2
4.3. Terrain API Examples	4-2
4.4. Changes to Network Representation of DtSimTasks	4-3
4.5. Miscellaneous API Changes	4-4
4.6. GUI API Migration Instructions	4-4
4.6.1. Removed Classes	4-4
4.6.2. New Classes	4-5
4.6.3. The Names of Factory Classes have Changed	4-5
4.6.4. Changes to Existing Classes	4-5
4.6.5. Changes to the Menu System	4-7
4.6.6. Event Controllers and Event Handlers	4-16
4.6.7. The <i>DtEntityInfoDialog</i> Class	4-17
4.6.8. Editing Plan View Display Examples for Use with VR-Forces	4-18
4.7. New Architecture for Connecting Components	4-18
4.7.1. Migrating to the New Port/Port Group Architecture	4-19
4.7.2. Creating Ports	4-19
4.7.3. Changes to Components that Send Data through Ports	4-21
4.7.4. Changes to Components that Receive Data through Ports and Port Interfaces	4-21
4.7.5. Updating Object Parameters Database Entries	4-22

Chapter 5 Migrating from 3.4 to 3.5

5.1. GUI API Migration Instructions	5-2
5.1.1. Removed Classes	5-2
5.1.2. New Classes	5-3
5.1.3. Changes to Existing Classes	5-4
5.1.4. Migrating Code for Subclasses of <i>DtMtlSymbolMapper</i>	5-4
5.1.5. Migrating Code from <i>DtLineConfig</i> and other Config Widgets	5-4
5.1.6. Migrating Code that Added Coordinate Systems or Unit Converters to Dialog Boxes	5-5
5.1.7. Updates to <i>vrfsim.opd</i> for Overlay Support	5-5
5.2. Support for Large Terrain Databases	5-8
5.2.1. Loading Geocentric Terrain Databases in VR-Forces	5-8
5.2.2. Geocentric Terrain Databases	5-9
5.2.3. Changes to the Terrain API to Support Large Terrain Databases	5-9
5.2.4. Other API Changes	5-10
5.2.5. Migrating Your Code to Support Large Databases	5-12
5.3. Migrating to the Updated Terrain API	5-13
5.3.1. New Manager Classes	5-13
5.3.2. Migrating from List Nodes to the New Manager Classes	5-14
5.3.3. Creating and Adding <i>DtPolygonIndirects</i> and <i>DtTriangleIndirects</i>	5-14
5.3.4. New Code Examples	5-16
5.3.5. Memory Usage	5-16
5.3.6. Optimizing Performance	5-17
5.3.7. Deprecated Classes	5-18

Chapter 6 Migrating from 3.5 to 3.6

6.1. Changes to the GUI API	6-2
6.1.1. Additions to the GUI API	6-3
6.2. Changes to how Paper Maps are Drawn	6-5
6.3. Terrain API and tdbtool Changes	6-5
6.3.1. Clipping Vector Data to the Terrain Extent	6-5
6.3.2. Planarizing Vector Data	6-6
6.3.3. tdbtool Options	6-6
6.3.4. New Map File Options	6-6
6.3.5. <i>DtTerrainDatabase</i> Functions	6-7
6.3.6. The geometryNative Library has been Deprecated	6-7
6.4. Miscellaneous API Changes	6-8

Chapter 7 Migrating from 3.6 to 3.7

7.1. The Object Parameters Database API and File Format have Changed	7-2
---	-----

7.1.1. New Object Parameters Database Format	7-2
7.1.2. Updating Legacy Object Parameters Database Files	7-3
7.1.3. New Object Parameters Database API	7-5
7.2. Ground Entities Avoid Obstacles and Features	7-5
7.2.1. Configuring Obstruction Avoidance	7-6
7.3. Saving Default Entity Parameters to the Order of Battle File	7-8
7.4. New Examples	7-8
7.4.1. convertParamDb	7-8
7.4.2. createParamDbDynamically	7-9
7.4.3. condition	7-9
7.4.4. showMissileImpact	7-9
7.5. Collision Detection and Avoidance Has Been Rewritten	7-9
7.5.1. Component Changes	7-9
7.5.2. API Additions	7-10
7.6. Updates to the Move-To Controller Component Hierarchy	7-10
7.7. Changes to the <i>DtGroundMoveToControllerComponent</i> API	7-11
7.7.1. Migration to the New MoveTo API	7-11
7.8. Changes to the Terrain API	7-12
7.8.1. Deprecated Terrain API Classes	7-12
7.8.2. Updates to the <i>DtVectorNetwork</i>	7-12
7.8.3. Clarification of Vector Network Intersection	7-13
7.8.4. Updates to Metrics Used to Calculate Intersections	7-13
7.8.5. Updates to the Display of Vector Point Features	7-13
7.9. GUI API Changes	7-13
7.9.1. <i>Ability to Select Terrain Feature Objects</i>	7-14
7.9.2. The Entity Information Dialog Boxes have been Redesigned	7-14
7.9.3. Creating Graphical Objects	7-16
7.9.4. Setting Resources	7-17
7.9.5. Filtering Pick Lists	7-17
7.9.6. Other Changes to the GUI API	7-19
7.10. VR-Forces API changes	7-20
7.10.1. Iterators	7-20
7.10.2. Target Detection Sensor	7-20
7.11. Miscellaneous API Changes	7-24
7.12. New Build System	7-27

Chapter 8 Migrating from 3.7 to 3.7.1

8.1. New Examples	8-2
8.2. New Configuration Parameters	8-2
8.3. Support for Cruise Missiles	8-2
8.3.1. Changes to the VR-Forces API to Support Cruise Missiles	8-3
8.4. Copying Parameter Entries	8-6
8.5. API Changes	8-7
8.6. Updating Object Parameters Databases	8-7

Chapter 9 Migrating from 3.7.1 to 3.8

9.1. Object Parameters Database Editor	9-2
9.2. Configuration and Startup Changes	9-2
9.2.1. Changes to Pathname Interpretation in Object Parameters Database Files	9-2
9.2.2. Configuring Search Paths in Scenario Files	9-2
9.2.3. New Startup Options	9-3
9.2.4. Default Scenario	9-4
9.2.5. Configuration File Changes	9-4
9.2.6. Host Address String	9-6
9.2.7. Temporary Directories	9-6
9.2.8. Changing or Hiding the Background Character	9-6
9.3. Entity Modeling	9-6
9.3.1. Path Planning	9-7
9.3.2. Road Driving	9-8
9.3.3. Improved Aggregate Behaviors	9-9
9.4. Terrain Database Changes	9-10
9.4.1. Flat Earth Support	9-10
9.4.2. RPF Resolution Keyword Addition	9-11
9.5. Component Code Generator	9-11
9.6. Improved Performance	9-11
9.7. New GUI Features	9-11
9.7.1. LOS Filter	9-11
9.7.2. Reflection of the Map View	9-11
9.7.3. Freehand drawing	9-11
9.7.4. Storing Line Settings	9-12
9.7.5. Specifying Line Width	9-12
9.7.6. Entity Capabilities	9-12
9.7.7. Setting Appearance Attributes	9-12
9.7.8. Loading the Create Menu Dynamically	9-13
9.7.9. Ability to Create Cultural Features	9-13
9.7.10. Transmitter and Signal Support	9-13
9.7.11. Threat Circles (Range Rings)	9-13
9.7.12. Object Transparency	9-14
9.7.13. Replacing Entity Icons with Generic Symbols	9-14
9.7.14. Papermap Transparency	9-14
9.7.15. Stealth Navigation Toolbar	9-14
9.7.16. Displaying Information About Environmental Objects	9-14
9.7.17. Fire/Detonate Option Separation	9-15
9.7.18. Enabling and Disabling the Display of Fire and Detonate Interactions	9-15
9.7.19. Hazard Cloud Display	9-15
9.7.20. Shadow Text	9-15
9.7.21. Ability to Specify A Symbol Map and Create Menu in Scenario Files	9-15

9.8. API Changes	9-15
9.8.1. GUI Remote Control	9-16
9.8.2. Loading Script Files at Startup	9-18
9.8.3. Object Transparency	9-18
9.8.4. API Changes to Support Transmitters	9-18
9.8.5. Symbol Mappers	9-19
9.8.6. Classed Removed	9-19
9.8.7. Changes to the FED File	9-19
9.8.8. Object Parameters Database Changes	9-20
9.8.9. Centralized Fire and Detonation Processing	9-22
9.8.10. <i>DtBoundingVolume</i> Migration Guide	9-23
9.8.11. Changes to the <i>DtCgf</i> Class	9-28
9.8.12. Spatial Organization of Objects	9-29
9.8.13. Spatial Subdivisions	9-31
9.8.14. Symbol Strings	9-36
9.8.15. New Classes for Managing Aggregates	9-37
9.8.16. Ability to Extend State Repositories at the Base Class Level .	9-39
9.8.17. Terrain API Changes	9-40
9.8.18. Miscellaneous VR-Forces API Changes	9-42
9.8.19. New Examples	9-48

Chapter 10 Migrating from 3.8 to 3.9

10.1. GUI API Changes	10-2
10.1.1. Showing and Hiding Symbols	10-2
10.1.2. Overlay View Manager	10-3
10.1.3. New View Menu Action Groups	10-4
10.1.4. Help Menu	10-4
10.1.5. Changes Made to Implement the Objects Toolbar	10-4
10.1.6. Tdb and Papermap Drawers	10-6
10.1.7. <i>DtTerrainLocationEntryWidget</i>	10-6
10.1.8. Object Mapping Classes	10-6
10.1.9. Setting Font Size on Linux	10-6
10.1.10. Overlay Objects can be Considered Control Objects	10-7
10.1.11. <i>DtNameLabelSymbol</i> Adds Layer Member	10-7
10.1.12. <i>dropEvent()</i> Renamed	10-7
10.1.13. Terrain Selection Changed	10-7
10.1.14. Changes to <i>DtTerrainLineConfigWidget</i>	10-7
10.2. Terrain API	10-8
10.2.1. Openflight Reader	10-8
10.2.2. MetaFlight Reader	10-9
10.2.3. Changes to Shapefile Support	10-9
10.2.4. Changes to GDB	10-9
10.2.5. <i>FltMetafileRecord</i> and <i>DfdDfadMetafileRecord</i> Additions	10-10
10.2.6. <i>DtExtentInitializer</i> Respects Z Values	10-10

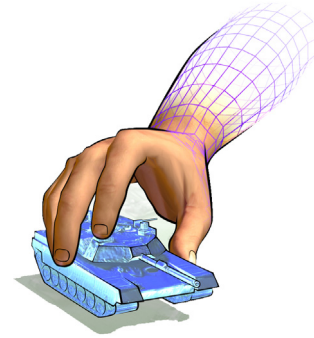
10.2.7. GDB File Format Changes	10-11
10.2.8. <i>DtGdbNode</i> Class Hierarchy Interface Changes	10-11
10.2.9. <i>DtTerrainDatabase</i> Interface Changes	10-12
10.2.10. Coordinate System Class Hierarchy Changes	10-12
10.2.11. Library Changes	10-12
10.3. Simulation API Changes	10-13
10.3.1. The Physical World	10-13
10.3.2. Object Parameters Database and Related Configuration Changes	10-13
10.3.3. Embarkation Controller	10-17
10.3.4. Attaching Environmental Objects to Entities	10-20
10.3.5. Initializing Extended State Repositories	10-20
10.3.6. Assume Tasks Have Been Removed	10-21
10.3.7. Changes Supporting Plan Language Improvements	10-22
10.3.8. New and Changed Component Classes	10-22
10.3.9. Handling Formation-changed Events in Aggregate Move-along and Patrol-Along Controllers	10-26
10.3.10. <i>DtVrfObjectPredicate</i> Changes	10-26
10.3.11. Determining when an Object's Vertices Change	10-26
10.3.12. <i>DtFormationTable</i> can be Overridden	10-27
10.3.13. Support for Simulation Management PDUs in DIS	10-27
10.3.14. New, Changed, and Deprecated Examples	10-28
10.3.15. Miscellaneous API Changes	10-29

Chapter 11 Migrating from 3.9 to 3.10

11.1. API Changes	11-2
11.1.1. New Plug-In API Architecture	11-2
11.1.2. Changes to the Rotary Wing Model	11-2
11.1.3. Changes to Registration for Task and Set Data Request Messages	11-4
11.1.4. Simulation API Changes	11-5
11.1.5. GUI API Changes	11-5

Chapter 12 Migrating from 3.10 to 3.11

12.1. API Changes	12-2
12.1.1. Human Movement Model	12-2
12.1.2. Entity-Based Plan Management	12-2
12.1.3. New Communications Mechanism	12-2
12.1.4. Support for Plug-ins in the OPD Editor	12-2
12.1.5. Entities Share Detection Lists	12-3
12.1.6. Versioning of VR-Forces Messages	12-3
12.1.7. Improved API Class Documentation	12-3
12.1.8. Upgrade to Qt 4.3.3	12-3
12.1.9. Miscellaneous API Changes	12-7



Preface

This manual is for developers who must migrate VR-Forces applications from a legacy release to a more recent release. It contains the cumulative API release notes for the VR-Forces 3.x series of releases. Each chapter describes migration issues for one release upgrade.

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



This manual does not cover migrating to the most recent release of VR-Forces from the previous release. Please see *VR-Forces Release Notes* for the most current migration information.

VR-Forces Documentation

The VR-Forces documentation set is as follows:

- ♦ *VR-Forces Getting Started Guide* is a quick introduction to VR-Forces. It covers the basics of installing VR-Forces, running a scenario, and creating a scenario. It focuses on helping new users avoid common mistakes.
- ♦ *VR-Forces Users Guide* describes how to install and use the VR-Forces program. It is distributed in printed format and as an online manual in Adobe Portable Document Format (PDF).
- ♦ *VR-Forces 3D Front End Users Guide* explains how to use the 3D features in the VR-Forces 3D Front End. Configuration of the 3D Front End is covered in *VR-Forces Configuration Guide*. It is distributed in printed format and in PDF format.
- ♦ *VR-Forces Configuration Guide* explains advanced features for configuring performance and how to edit VR-Forces configuration files. It includes documentation of the Entity Editor, OPD Editor, and Scenario Merge tools. It is distributed in printed format and in PDF format.
- ♦ *VR-Forces Developers Guide* explains how to use the VR-Forces APIs. It is distributed in printed format and in PDF format.
- ♦ *VR-Vantage Developer's Guide* describes the VR-Vantage API. This is the API used to build the VR-Forces 3D Front End. Developers who want to customize or extend the 3D Front End will need to refer to this manual.
- ♦ *VR-Forces Master Index* combines the index from the VR-Forces manuals, including TDB Tool, to help you find references to subjects that are covered in two or more manuals or locate the manual in which a particular topic is discussed. It is distributed only in PDF format.
- ♦ *VR-Forces Migration Guide* collates API migration information for each release in the VR-Forces 3.x series. It is distributed only in PDF format.
- ♦ Online help. The VR-Forces GUI, the OPD Editor, and the TDB Tool have online help accessible from the Help menu.
- ♦ Class documentation. Classes are documented in HTML.
- ♦ *VR-Forces Release Notes*. Release documentation consists of release notes provided as a PDF file and in printed format.

The PDF files are in the *./doc* directory.

Online Manuals

The online manuals (PDF) use hypertext links for the entries in the table of contents, for all index entries, and for cross references. PDF documentation also supports full text searching.

To view and print PDF documents, you need a copy of the free Acrobat Reader software (version 5.0 or later). To obtain a copy, visit www.adobe.com.

MÄK Products

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

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

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

- **MÄK RTI.** An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK RTI is optimized for high performance. It has an API, RTIsby®, that allows you to extend the RTI using plug-in modules. It also has a graphical user interface (GUI) for the MÄK RTI rtiexec.
- **VR-Forces®.** VR-Forces is a computer generated forces application and toolkit. It provides an application with a GUI, that gives you a two dimensional view of a terrain database.

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

- **VR-Vantage™.** VR-Vantage is your 3D visualization solution. It includes two end-user applications, MÄK Stealth and Vantage IG, and the VR-Vantage Toolkit. The Stealth displays a realistic, 3D view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway.

Vantage IG is a configurable desktop image generator for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.

The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend Stealth and Vantage IG or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.

- ♦ **MÄK Data Logger.** The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- ♦ **MÄK Plan View Display.** The Plan View Display (PVD) provides a 2D, “big picture” of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display.
- ♦ **VR-Exchange™.** VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and limited DIS translation.
- ♦ **MÄK Gateway.** The Gateway is a translator application that allows DIS exercises to operate as members of HLA federation executions that use the RPR FOM, thereby preserving the value of simulation code written to support DIS. The MÄK Gateway is a standalone application and does not require you to write additional code for its use.

How to Contact Us

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

Telephone

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

E-mail

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

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/licenses.php
Product version and platform information:	www.mak.com/support/productversions.php
For the free, unlicensed MÄK RTI:	www.mak.com/products/rti.php
Support FAQ:	www.mak.com/support/faq.php

Post

Send postal correspondence to:	VT MÄK 68 Moulton St. Cambridge, MA, USA 02138
--------------------------------	--

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

Document Conventions

This manual uses the following typographic conventions:

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

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

Hardware and Operating System Naming Conventions

Unless otherwise specified, references to UNIX include Linux, regardless of the type of computer it is running on. References to PCs refer to Intel processor-compatible computers running a version of Microsoft Windows.

Mouse Button Naming Conventions

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

Third Party Licenses

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

Boost License

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

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

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

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

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

libXML and libICONV

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

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

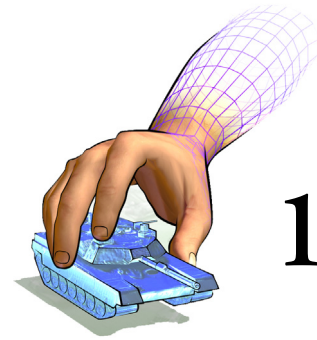
pThreads Library

VR-Exchange links with the pThreads win32 library. The library is distributed under the GNU Lesser General Public License (LGPL). MÄK has made no modification to this library. For information about the pThreads win32 library please see: <http://sourceware.org/pthreads-win32/>

EHS, PCRE, and PME

The MÄK RTI links with the EHS, PCRE, and PME libraries. These libraries are distributed under the GNU Lesser General Public License (LGPL). MÄK has made modification to these libraries only to allow them to compile on the supported platforms. For more information about these libraries please see the following web sites:

- ♦ EHS: <http://xaxxon.slackworks.com/ehs/>
- ♦ PCRE: <http://www.pcre.org/>
- ♦ PME: <http://xaxxon.slackworks.com/pme/index.html>



Migrating from 2.x to 3.0

This chapter contains the instructions for moving from VR-Forces 2.x to VR-Forces 3.0.

1.1. Updated Simulation API

VR-Forces 3.0 includes a variety of improvements to the top-level VR-Forces simulation API to make it easier to embed VR-Forces functionality into other applications, or customize or extend the simulation engine. Most of the changes are related to how applications initialize VR-Forces and instantiate the objects that manage the simulation as a whole. Chapter 1 of *VR-Forces Developer's Guide* and the updated examples in `./examples` provide details of the new classes you need to use to initialize and manage VR-Forces-based applications. Lower-level implementation classes like those that implement components, plans, parameters, messages, tasks, and so on, have remained largely unchanged, and there is a high degree of backwards compatibility with previous releases.

For those upgrading from pre-3.0 versions of VR-Forces, it is no longer necessary to instantiate *DtVrfCreator* or *DtSimManager* within your VR-Forces applications. Those classes still exist, but their importance has diminished. Their APIs and purposes have changed significantly, so changes to application code will be necessary. It is also no longer necessary to subclass *DtVrfCreator* to register custom subclasses of VR-Forces classes like *DtSimComponent*, *DtSimTask*, and so on, with VR-Forces.

Under the new scheme, a VR-Forces-based application instantiates the new *DtCgf* class to create a simulation engine object, and then ticks() that *DtCgf* object each frame to cause the simulation to occur. *DtVrfCreator* is now used only within *DtCgf* to initialize some top-level manager classes like *DtVrfObjectManager*, *DtPlanManager*, and so on. *DtSimManager* is still used by *DtCgf* to schedule events and timers, but many applications will never need to know that *DtSimManager* even exists. The new method for registering custom subclasses for components, tasks, messages, and so on, is to register a creator function with the appropriate factory. Factories are accessible through *DtCgf*'s *DtFactoryManager*.

The code for the `vrfSim` application and other API examples has been greatly simplified. They now use a class called *DtVrfApp* to handle command-line options, and initialize and tick the *DtCgf* object for you. We include source code for *DtVrfApp* as a model for you in creating your own applications or extending `vrfSim`.

Again, see *VR-Forces Developer's Guide* and `./examples/*` for details.

1.2. Changes to the GUI/Simulation Engine Communications

In VR-Forces 2.x, the vrfGui application played a larger role in loading and initializing scenarios. The front-end would load a scenario file, and send individual messages to relevant simulation engines asking them to instantiate each entity in turn, assign plans, and so on. In VR-Forces 3.0, the job of loading scenarios is handled mainly in the simulation engines. When you choose Load Scenario in the GUI (or call `loadScenario()` from the Remote Control API), the front-end merely breaks the order of battle and plan files into sub-files describing the simulation responsibilities of each simulation engine. It then sends those subfiles to the relevant back-ends, where they are read by the simulation engines. Saving scenarios works similarly. Simulation engines save individual OOB and plan files, and send those files to the control application (for example, vrfGui) that asked for the save. That control application merges the files sent by the various back-ends, and saves the resulting OOB and Plan files for the scenario as a whole.

1.3. Improved Performance

VR-Forces has made significant changes to the simulation engine to improve performance. It also now includes a variable tick rate parameter that you can set in the object parameters database on a per-entity or per-component basis. For information about setting the variable tick rate and other ways to enhance performance, see Chapter 10, “Advanced Configuration”, in *VR-Forces Migration Guide*.

VR-Forces’ performance will vary based on the following:

- ♦ Size of the terrain database
- ♦ Amount of memory available on your computer
- ♦ Computer speed and processing power
- ♦ Number of objects (entities, control objects) in the scenario and how active their plans are
- ♦ The number of objects you try to simulate on a given back-end
- ♦ The frequency of the tick rate that you set
- ♦ The number of front-ends and back-ends you run on your computer.

1.4. Miscellaneous Toolkit Changes

This section highlights changes to the toolkit that you should be aware of:

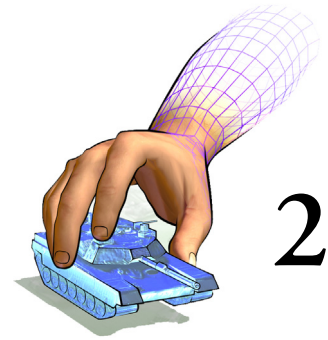
- ♦ The Status Manager has been replaced by the Status Publisher in the back-end and the Status Listener in the front-end. VR-Forces back-ends no longer need to be aware of other back-ends. A back-end now publishes its presence to the network using the *DtVrfStatusPublisher* class. The GUI learns about VR-Forces back-ends through its *DtVrfStatusListener* class.
- ♦ *DtVrfMessageInterface* (*vrfMsgIf.h*) replaces the *DtSimManagerMessageInterface* class.

1.5. Remote Control API

The VR-Forces Remote Control API is a set of classes that allow an application to easily control one or more remote VR-Forces applications. The MÄK VR-Forces GUI (the *vrfGui* executable) uses the VR-Forces Remote Control API to control the *vrfSim* application or other VR-Forces applications. The Remote Control API can also be used in custom VR-Forces front-ends, simulation managers, or instructor/operator stations.

The Remote Control API contains no GUI-related code. You can use it as easily in an application that uses scripts or command-line input as you can in a GUI-based application. You can use it in any application that needs to control remote VR-Forces simulation engines.

VR-Forces Developer's Guide describes the remote control API.



Migrating from 3.0 to 3.1

This chapter contains the instructions for moving from VR-Forces 3.0 to VR-Forces 3.1.

2.1. The VR-Forces GUI API

The VR-Forces GUI API enables you to build custom front-ends for viewing and controlling VR-Forces simulation back-ends.

The VR-Forces GUI API is built on top of Qt, a multi-platform framework for building graphical user interfaces. It is also layered on top of VR-Link, which provides the DIS/HLA networking interface. It uses the VR-Forces remote control API to control remote VR-Forces simulation back-ends. The remote control API is described in Chapter 10 of *VR-Forces Developer's Guide*.

2.2. Overview of the Main Classes in the GUI API

This section describes the key classes in the VR-Forces GUI API.

2.2.1. The Top-Level Classes

- ♦ *DtVrfGuiApplication* (*vrfGuiApplication.h*) is the top-level application class. The application contains the main event loop. It uses a *DtVrfGuiFactory* to create various kinds of top-level objects that it owns, such as the main window, the VR-Forces remote controller, the Terrain Database Manager, and the DIS/RPR Connection Manager. There is only one instance of the *DtVrfGuiApplication* class in the application. It inherits from the Qt class, *QApplication*.
- ♦ *DtVrfGuiFactory* (*vrfGuiFactory.h*) is the factory for generating many of the important objects in the application, for example, the main window and the terrain map area. You can override the factory to create derived types of objects, to build custom applications.
- ♦ *DtVrfGuiWindow* (*vrfGuiWindow.h*) is the main window in a *DtVrfGuiApplication*. It contains the menu bar, the terrain map, the mini map, the echelon view dialog, and the other toolbars and dialog boxes. It inherits from the Qt class, *QMainWindow*.
- ♦ *DtVrfRemoteController* (*vrfController.h*) is actually part of the VR-Forces remote control API. The *DtVrfGuiApplication* uses the remote controller to learn about remote VR-Forces back-ends and to control them. Through the remote controller, the GUI is able to do things such as create, load and save scenarios, create entities and control objects, task entities, and create plans. The remote control API is documented in MÄK VR-Forces Developer's Guide.
- ♦ *DtTdbManager* (*tdbManager.h*) manages the loading of terrain databases and paper maps.
- ♦ *DtConnectionManager* (*connectionManager.h*) manages the DIS/HLA network connection. It contains the VR-Link lists of reflected entities, aggregates, and environmental processes.

2.2.2. Other Important Classes

- *DtVrfGuiMapArea* (*vrfGuiMapArea.h*) is the primary terrain map display in the main window (*DtVrfGuiWindow*). It is responsible for the drawing and navigation of terrain. It inherits from the Qt class, *QWidget*.
- *DtTerrainMiniMap* (*tmMiniMap.h*) is the miniature display of terrain that provides the global view of the terrain. It is owned by the *DtVrfGuiWindow*. It inherits from the Qt class, *QWidget*.
- *DtModelData* (*modelData.h*) provides the non-graphical representation of the various kinds of simulated objects in *vrfGui*, including entities, aggregates, control objects, and fire and detonation interactions. These objects are maintained by the *DtVrfGuiApplication*.
- *DtPvdSymbolGenerator* (*pvdSymbolGenerator.h*) generates the appropriate *DtModelData* objects from the network objects (entities, aggregates, control objects, and fire and detonation interactions) received over the network through the *DtConnectionManager*. For example, whenever a new remote entity is added to the Connection Manager, the *DtPvdSymbolGenerator* creates a corresponding *DtModelData* item to represent it. It manages the addition, updating, and removal of the *DtModelData* objects for the application. It inherits from the Qt class, *QObject*.
- *DtSymbol* (*symbol.h*) is the base class for all drawable symbols (for example, bitmaps, circles, lines, etc.). *DtSymbols* are drawn using a *DtSymbolDrawer*. It inherits from the Qt class, *QObject*.
- *DtSymbolDrawer* (*symbolDrawer.h*) is the abstract base class for all symbol drawers. Symbol drawers know how to draw symbols in a particular style (for example, filled, outline, pattern filled, pen thickness). It inherits from the Qt class, *QObject*.
- *DtViewDriver* (*viewDriver.h*) manages the symbols inside a view. The *DtVrfGuiMapArea* and *DtTerrainMiniMap* each have their own *DtViewDriver* class that manages the symbols in their displays. The view driver owns the *DtSymbol* list for a view. It has slots for adding, removing, modifying, and updating symbols. It inherits from the Qt class, *QObject*.
- *DtMtlSymbolMapper* (*mtlSymbolMapper.h*) maps the non-graphical simulated objects (*DtModelData* objects) to the corresponding drawable *DtSymbol* object(s) to be drawn on the terrain map). It is used to map the simulated entities, aggregates, control objects, and interactions to drawable symbols. The *DtVrfGuiApplication* has a *DtMtlSymbolMapper*. While there is only one symbol generator creating the single set of model data objects shared by all views in the application, each terrain map display (and minimap display) has its own instance of a symbol mapper. This is because different views may want to visualize the same model data objects using different symbols. It inherits from *QObject*.
- *DtPvdBlipSymbolMapper* (*pvdBlipSymbolMapper.h*) is similar to the *DtMtlSymbolMapper*, it performs the *DtModelData* to *DtSymbol* mapping for the Minimap. The *DtVrfGuiApplication* has a *DtPvdBlipSymbolMapper*. It inherits from the Qt class, *QObject*.
- *DtFilterLens* (*filterLens.h*) is the base class for all filter lenses. A filter lens is an

object that is used to alter the display of terrain and symbols in map areas. For example, filter lenses can be used to filter out objects you don't want to see, change the appearance of icons, or change the appearance of terrain. Different types of filters are implemented in derived classes.

Filter lenses can either be windows that overlay a *DtLensArea*, or non-visual items that change symbol attributes in a *DtLensArea*. An important derived kind of *DtFilterLens* is *DtSymbolFilterLens*. These filters apply to *DtSymbols* (the drawable items in the terrain map). Classes derived from *DtSymbolFilterLens* are used to affect how symbols are displayed. For example, the derived *DtForceSymbolFilter* class (*forceSymbolFilter*), shows or hides the symbols of a particular DIS/RPR force type. It inherits from the Qt class, *QWidget*.

- ♦ *DtLensArea* (*lensArea.h*) represents an area in which you can apply any number of filters to modify its appearance. It maintains a set of *DtFilterLens* objects. It defines signals and slots for managing them. An important type of derived *DtLensArea* is *DtSymbolLensArea* (*symbolLensArea*), specifically meant to handle *DtSymbolFilterLens* objects. The *DtVrfGuiMapArea* has-a *DtSymbolLensArea* area. This enables any number of *DtSymbolFilters* to be added to the terrain map, to affect the appearance of symbols. The *DtVrfGuiMapArea* comes configured with a filter in its lens area for resizing icons (*DtPvdResizeSymbolLens*). The Fog of War example demonstrates how to create and add a new type of *DtSymbolFilter* to the terrain map's lens area. It inherits from the Qt class, *QDialog*.

2.2.3. Customizing or Extending the VR-Force GUI Application

The *DtVrfGuiApplication* class is the top-level class you need to create in a custom application. In the sample application project we deliver (in *./appsrc/vrfGui*), we create an instance of *DtVrfGuiApplication* in *main.cxx*.

```
#include "vrfGuiFactory.h"
#include "vrfGuiApplication.h"

int main(int argc, char* argv[])
{
    DtVrfGuiFactory f;

    DtVrfGuiApplication a(argc, argv);

    if (!a.init(&f))
    {
        return -1;
    }

    a.fileNewWindow();

    return a.exec();
}
```

DtVrfGuiApplication takes command line arguments in its constructor. It caches them for processing in its *init()* function. The call to the application's *init()* member is where the application gets initialized. In *main.cxx* we do the following:

A *DtVrfGuiFactory* is passed into `init()`, which the *DtVrfGuiApplication* class uses to drive the construction of many of the objects used in the application.

The call to the application's `fileNewWindow()` function creates the main window (a *DtVrfGuiWindow*).

The application's `exec()` function is called, which starts the main event loop for the application.

In many cases, you will not need to subclass the application class or the factory class to override the specific parts of the VR-Forces GUI application you are interested in. You just need to configure the factory (through member function calls) to change the kinds of objects the factory creates for the application. For example, in the Fog of War example, we only want to create a derived kind of main window, so we can configure it with a new kind of symbol filter. To do this, we just have to make a single function call to the factory, telling it to create our new kind of main window class, instead of the default, for example:

```
DtVrfGuiFactory factory;

// Configure the factory so that it creates our new kind of main
// window.
factory.setMainWindowCreatorFcn(MyMainWindow::create);

DtVrfGuiApplication app(argc, argv);

if (!app.init(&factory))
{
    return -1;
}
```

The `MyMainWindow::create()` function is a static function we defined in our new main window class that returns a new instance of the class.

```
DtTMWindow* MyMainWindow::create(QWidget* parent, const char* name,
    WFlags flags)
{
    return new MyMainWindow(parent, name, flags);
}
```

By setting a different main window creator function in the factory, whenever the application requests a new main window object from the factory, the factory will use our new create function to return our derived type of window.

You can use this approach to overriding the factory for many of the high-level objects used in the VR-Forces GUI API (for example, the main window, terrain map area, the connection manager, and so on).

2.2.4. Miscellaneous Qt Issues

This section describes items you should keep in mind when working with Qt and the VR-Forces GUI API.

Using Forms with Qt Designer

Qt Designer is a Qt application for designing and building user interfaces interactively. In Qt Designer, you can create custom widgets and save the results to file. The files saved by Qt Designer (.ui files) are referred to as forms. These files are XML files that you compile to source code using the Qt User Interface Compiler (UIC). The UIC takes as input a form file, and generates C++ header and source files as output. The resulting source files can then be compiled and linked into an application. We do not deliver forms as part of the VR-Forces Toolkit. We only deliver the source code generated from them by the Qt User Interface Compiler. Therefore, you have to subclass our classes to extend their functionality. However, you can still use Qt Designer to build new dialog boxes and windows, and integrate them into the VR-Forces GUI application. The Add Task example shows how to do this, demonstrating how to add a completely new task dialog box (created using Qt Designer) to the VR-Forces GUI application.

Using Signals and Slots

Qt provides a mechanism for communicating between objects, known as signals and slots. Under this scheme, an object can emit a signal, representing some event, and any number of receiving objects can receive the signal if they have been connected to the signal through slots (functions to handle the signal). It is a flexible alternative to traditional callback methods. The VR-Forces GUI API uses the Qt signal/slot mechanism for communicating between objects. We define numerous signal and slots in the VR-Forces GUI API. If you plan to use this capability in your derived classes, keep in mind the following:

- Only those classes derived from the Qt class `QObject` can use signals and slots.
- You must include the `Q_OBJECT` macro in the class definition of derived classes that contain signals and slots. Based on Qt documentation, we recommend that you include it in all classes derived from `QObject`.

Custom Compilation Steps

We do not deliver Qt's qmake project files (.pro) for building the sample application and examples. We deliver Microsoft Visual C++ project files (.dsp) and UNIX Makefiles. You can develop qmake project files for your applications (see the Qt documentation for more information). However, you can also extend the project files we deliver with the VR-Forces Toolkit.

Compiling Qt Designer Forms

In Windows, if you add new forms (.ui files) to your Microsoft Visual C++ project file, you must run the Qt User Interface Compiler (UIC). Consult the Qt documentation for details.

Running the Meta Object Compiler

For any new classes that you create that use signals and slots, you must run the Meta Object Compiler (MOC). The MOC reads the source file and generates new source code that must be compiled and linked into your application. Consult the Qt documentation for details.

2.3. Building the vrfGui Application and Examples

We deliver a project and Makefile for building the vrfGui application in `./appsrc/vrfGui`. The VR-Forces GUI examples are included in the `./examples/examples.dsw` workspace (combined with all of the VR-Forces back-end example projects).

2.4. Introduction to the GUI API Examples

The VR-Forces GUI API includes four examples (in the `./examples` directory):

- ♦ `fogOfWar` – This example shows how to modify the display of entity icons in the terrain map, simulating the effects of fog of war. Normally, the GUI displays all entities (friendly, hostile, other) in a simulation. In this example, it limits the display to just the friendly (blue) entities, and only those non-friendly force entities within a certain range of the blue force.
- ♦ `addTaskGui` – This example shows how to add a new task to the VR-Force front-end. In particular, it shows how to add a new dialog box for configuring the new task (a retreat task), how to hook it up to the appropriate menus, and how to send the new task to an entity through the remote control API. It works in conjunction with the `addTaskSim` example (formerly the `userTask` example), which adds the capability to simulate the behavior needed to carry out the retreat task in the back-end.
- ♦ `simpleScenarioPlayer` – This example demonstrates how to alter the appearance and functionality of the VR-Forces GUI by removing various buttons, menu items, and so on. from the interface. It transforms the standard VR-Forces GUI interface into a simple scenario player that only allows a user to load and play scenarios.
- ♦ `drawMissileImpact` – This example shows how to draw additional information on the display. Specifically, it shows how to draw an additional graphical item (an ellipse) on the terrain map display, whenever a detonation is received.

2.4.1. The Fog of War Example

Source code and project files for this example are in `./examples/fogOfWar`.

In this example, we want to show the effects of fog of war in the terrain map. Instead of having complete knowledge of the locations of all entities in a simulation, we want to show the battlefield from the standpoint of a particular force (in this case the DIS/RPR friendly force). It limits the display of icons in the terrain map to only those that the friendly force should know about.

To do the kind of filtering we want to do in the terrain map, we need to create a new kind of *DtSymbolFilterLens* (*symbolFilterLens.h*). *DtSymbolFilterLens*'s are used to alter the appearance of symbols in a view, such as the terrain map. In this example, to achieve the effect of fog of war, we want to hide some of the symbols in the display.

In this example, we create a new kind of *DtSymbolFilterLens*, *MyFogOfWarFilter*. This new kind of filter shows all of the entity symbols representing a given DIS/RPR force type. It hides the entity symbols of all other force types, unless they fall within a specified range of this force. The key function we have to override in this class is the pure virtual *modifyConditional()* member function.

The *modifyConditional()* function modifies the symbol passed in if, and only if, the filter criteria applies to it. It should return true if the symbol was modified in the function (for example, show, hidden, highlighted, resized, and so on), false if not. This function is periodically invoked by the lens area to which it is attached.

For example, if we define a type of filter that merely hides friendly symbols and shows all others, then in the *modifyConditional()* function in this filter, we apply the following logic:

- If the symbol is a friendly icon, modify the symbol (hide it), and return true.
- If the symbol is a non-friendly symbol, don't modify the symbol, and return false.

When implementing a new kind of filter, it is up to you to implement the new filtering algorithm in *modifyConditional()*, adhering to rule that if you modify the symbol in this function you return true, and otherwise return false. Modifying the symbol can mean any number of things, for example, showing, hiding, highlighting, changing the color, resizing, and so on.

In our *MyFogOfWarFilter* implementation of *modifyConditional()*, we look up the *DtModelData* object associated with the symbol, check its force type, and then show or hide it according to our model. For all entity symbols that get passed in, we modify the symbol (showing or hiding it according to our algorithm), and return true (indicating the symbol was modified). For all non-entity symbols (symbols not modified by this filter in any way), we return false (indicating the symbol was not modified).

```
bool MyFogOfWarFilter::modifyConditional(DtSymbol* newSymbol)
{
    . . .

    DtEntityModelData* entity =
        dynamic_cast<DtEntityModelData*>(modelData);

    if(!entity)
    {
        // Model may not be an entity (for example, it may be an aggregate or
        // a control object)
        return false;
    }

    if(entity->getForceId() == myForceType)
    {
        // We know where our own forces are - show the symbol.
        newSymbol->show();
    }
}
```

```

        return DtTrue;
    }

    // Entity must be of some other force type. Check to see if is in range
    // of any of the entities in our force.
    if(inRange(*entity, myForceType, myRangeSqr))
    {
        // Entity of other force type is in range of our force - show the
        // symbol
        newSymbol->show();

        return DtTrue;
    }

    // Entity was not of our force type, and it was not in range of our
    // forces - hide the symbol.
    newSymbol->hide();

    return DtTrue;
}

```

The next thing we have to override is the main window class, so we can create this filter and add it to our terrain map whenever a new main window is created. To do this, we create a new *DtVrfGuiMainWindow*, and override its *init()* function:

```

bool MyMainWindow::init(const DtTMFactory* factory, DtTMMiniMap* miniMap)
{
    if(!DtVrfGuiWindow::init(factory, miniMap))
    {
        return false;
    }

    myFogOfWarFilter = new MyFogOfWarFilter(DtForceFriendly, areaMap(),
        "Lens", areaMap()->symbolLensArea(), tr("Fog of War"));

    myFogOfWarFilter->setRange(800.0);

    areaMap()->symbolLensArea()->addGlobalFilter(myFogOfWarFilter);

    return true;
}

```

Finally we need to tell the application to create our new kind of main window. See [“Customizing or Extending the VR-Force GUI Application,”](#) on page 2-4 for a discussion of how to do this.

2.4.2. The Add Task Example

Source code and project files for this example are in *./examples/addTaskGui*.

This example demonstrates how to add a new task dialog to the *vrfGui* application. When you add a new task to the VR-Forces back-end, to perform some new kind of action, it is likely that you will want to be able to configure and assign the task from the VR-Forces front-end. This example adds the dialog box for the retreat task. It works in conjunction with the *addTaskSim* example that adds the task to the back-end.

Adding the Task and Dialog Box

To create the new task and dialog box, we have to do the following:

- Create a new *DtSimTask*, to represent our new retreat task. See section 8.4, “Deriving a new Task from *DtSimTask*” in MÄK VR-Forces Developer’s Guide, for details about how to create a new task.
- Create a dialog box to configure the task using Qt Designer. Save the .ui file. In this example, we created the form *retreatTaskWidget.ui*.
- Add the .ui file to your project (see “Custom Compilation Steps,” on page 2-6). The Qt UIC (User Interface Compiler) generates source code from the form. In this example, the UIC-generated source files are *RetreatTaskWidgetUi.{h,cxx}*.
- Subclass the new class *RetreatTaskWidget*, derived from *RetreatTaskWidgetUi*. We want to create a derived class so that we always go back into QtDesigner to modify the .ui file (Once you edit the source code in *retreatTaskWidgetUi.{h,cxx}*, you cannot load it back into QtDesigner to modify it). Our new class uses signals and slots, so we include the `QOBJECT` macro at the beginning of our class definition.

Add the Dialog Box to the GUI

To include the new dialog box in the VR-Forces GUI so that it appears in the appropriate menus and behaves like the native dialog boxes, we have to do the following:

- Implement OK and Cancel actions in the *RetreatTaskWidget* class
- Override *DtTaskActionGroup* to add a new item to the task group (the group of task menu items).
- Override *DtVrfGuiMapArea* to add a slot that creates a new *DtRetreatTaskDialog* upon receipt of ‘activated’ signals.
- Override *DtVrfMainWindow* to:
 - Create the new kind of *DtTaskActionGroup* and configure it with a new Retreat-Task entry.
 - Connect the ‘activated’ signal from the Retreat Task *DtTaskActionGroup* item.
 - Add an action validator for the retreat task.

Implement the OK and Cancel Buttons

In the *RetreatTaskWidget* class, we need to implement the actions to be performed when the user clicks the OK and Cancel buttons.

The *taskAndClose()* function handles the OK case, in which we want to fill out a retreat task and send it.

```
void RetreatTaskWidget::taskAndClose()
{
    myMapArea->startNormalSelectionMode();

    QString rules = myRetreatTaskComboBox->currentText();
```

```
DtRetreatTask retreatTask;

retreatTask.setRetreatKind(rules.latin1());

// Always emit a taskActivated signal after
// you send a task.
emit taskActivated(retreatTask);

close();
}
```

In the `cancelAndClose()` function, emit the appropriate signal and `close()`:

```
void RetreatTaskWidget::cancelAndClose()
{
    myMapArea->startNormalSelectionMode();

    // Always emit a cancelActivated signal when
    // you cancel a task.
    emit cancelActivated();

    close();
}
```

Override the `init()` function to connect the OK button's 'clicked' signal to the `taskAndClose()` signal, and the Cancel button's 'clicked' signal to the `cancelAndClose` slots. These slots handle the 'OK' and 'Cancel' buttons in the dialog.

```
bool RetreatTaskWidget::init(DtVrfGuiMapArea* map)
{
    . . .

    connect
    (
        myOKButton,
        SIGNAL(clicked()),
        this,
        SLOT(taskAndClose())
    );

    connect
    (
        myCancelButton,
        SIGNAL(clicked()),
        this,
        SLOT(cancelAndClose())
    );

    return true;
}
```

Add the Task to the Menus

To add the new task to the menus, we need to add it to the *DtTaskActionGroup* class. *DtTaskActionGroup* is a *QActionGroup*, which represents a group of *QActions* that can be added as a single item to a menu or toolbar. The *DtTaskActionGroup* is the list of Task menu items that appear in the Task pull-down and right-click popup menus. To add a new task action item to the group, we created *MyTaskActionGroup*, a subclass of *DtTaskActionGroup*. In our derived class, we override *init()* to create and configure a new *QAction* item for the task entry in the group, and connect the 'activated' signal to a slot:

```
bool MyTaskActionGroup::init(DtVrfGuiMapArea* mapArea)
{
    if(!DtTaskActionGroup::init(mapArea))
    {
        return false;
    }

    myTaskRetreatAction = new QAction( this, "myTaskRetreatAction" );
    myTaskRetreatAction->setText( trUtf8( "Retreat..." ) );
    myTaskRetreatAction->setMenuText( trUtf8( "Retreat..." ) );
    myTaskRetreatAction->setToolTip( trUtf8( "Task selected entity/
        aggregate to retreat." ) );

    // Connect the actions.
    connect
    (
        myTaskRetreatAction,
        SIGNAL(activated()),
        this,
        SLOT(taskSelectionRetreat())
    );

    return true;
}
```

The slot is another function we define in this class, which returns an instance of our dialog box and displays it:

```
bool MyTaskActionGroup::taskSelectionRetreatTask()
{
    RetreatTaskWidget* w = new RetreatTaskWidget
    (
        myDialogParent,
        NULL,
        theTaskWidgetFlags
    );

    if (!w->init(myMapArea))
    {
        w->close();
    }

    prepWidget(w);
    w->show();

    return true;
}
```

Similarly, we need to override the *DtVrfGuiMapArea* class (*MyMapArea*) to respond to ‘activated’ signals by creating a new instance of *DtRetreatTaskDialog*. This is done in its *taskSelectionRetreatTask()* member.

Override DtVrfGuiWindow

We need to override the following member functions of the main window class, *DtVrfGuiWindow*:

- ♦ *initTaskActions()* – Create new *DtTaskRetreatAction* data member function.
- ♦ *initTaskActionValidators()* – Add the new *DtTaskRetreatAction* to a mapping of actions to validator functions.
- ♦ *initConnection()* – Connect the new *DtRetreatTaskAction* data member’s ‘activated’ signal to the *MyMapArea*’s *taskSelectionRetreatTask()* slot.

We also need to provide the validator function for *DtRetreatTaskDialog*. This is done in *processTaskRetreatValid()*.

Finally we need to tell the application to create our new kinds of main window and map area classes. See [“Customizing or Extending the VR-Force GUI Application,”](#) on page 2-4 for details.

2.4.3. The Simple Scenario Player Example

Source code and project files for this example are in *./examples/simpleScenarioPlayer*.

In the simple scenario player example, we want to simplify the interface to be ‘read-only’ scenario viewer, that is, the user cannot create anything new through the interface, only view existing scenarios.

In this example, we only need to override one class, the *DtVrfGuiWindow* class. In our derived class, we override the following functions to hide or alter existing items in the window:

- ♦ *initMenus()* – Clear the existing menu bar menus after the base class has created it, and then repopulate it with only the items we want,

```
if(!DtVrfGuiWindow::initMenus())
{
    return false;
}

// We could go through and remove each item in the menu bar, but it's
// easier to clear all of them, and just re-add the 2 menus that we
// need.
myMenuWidget->menuBar()->clear();

myMenuWidget->menuBar()->insertItem(tr("&File"), myFileMenu, -1, 0);

myMenuWidget->menuBar()->insertItem(tr("Si&mulation"), mySimMenu,
    -1, 3);

return true;
```

```
initFileMenu() – Clear the File menu of its existing items, and then
repopulate it with the few items we want:
DtVrfGuiWindow::initFileMenu();
```

```
myFileMenu->clear();
```

```
// Add just the items we want to appear in the File pulldown menu.
myFileLoadScenarioAction->addTo(myFileMenu);
myFileCloseScenarioAction->addTo(myFileMenu);
myFileExitAction->addTo(myFileMenu);
```

- updateMiddleClickMenu() – Clear the contents of the middle click menu (it allowed actions such as creating entities, which we want to remove from the interface).
- updateRightClickMenu() – Clear the contents of the popup menu (it allowed modification of the execution of a scenario, such as tasking entities).
- initToolbars() – Remove all of the toolbars (for example, the navigation toolbar).

To incorporate the new main window class into the application, we have to configure the factory with our new class. See [“Customizing or Extending the VR-Force GUI Application,”](#) on page 2-4 for details.

2.4.4. The Draw Missile Impact Example

Source code and project files for this example are in `./examples/drawMissileImpact`.



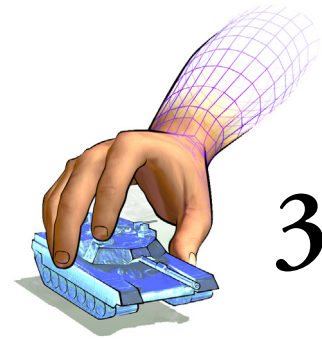
This example actually creates an instance of a *DtPvdApplication* (the MÄK Plan View Display). The VR-Forces GUI API is directly layered on the PVD API. We are including this example because everything in this example applies to the VR-Forces GUI API.

This example demonstrates how to draw an additional symbol whenever a DIS/RPR detonation interaction is received by the PVD. It draws a yellow ellipse at the location of a detonation (for example, the impact location of a missile) in the terrain map area, in addition to the standard MIL-STD 2525B detonate symbol that the PVD displays by default.

To add the new ellipse drawable to the display, we have to intercept the point at which the object representing the detonation interaction (received over the network as a DIS PDU or an HLA interaction) is mapped to a symbol to be drawn on the terrain display. This mapping occurs in the *DtMtlSymbolMapper* class (*mtlSymbolMapper.h*).

To add the new ellipse symbol, we create a derived *MyMtlSymbolMapper* class, and override the *DtMtlSymbolMapper* class's `createSymbol()` member function. This function takes a *DtModelData* object (the object in the PVD that provides the non-graphical representation of an object such as a detonation interaction, fire interaction, entity, or aggregate), and creates a corresponding *DtSymbol* to be displayed on the *DtPvdMapArea* (the main terrain map). The *DtSymbol* represents the drawable object you see on the terrain map.

To incorporate *DtMtSymbolMapper* class into the application, we have to configure the factory with our new class. See [“Customizing or Extending the VR-Force GUI Application,”](#) on page 2-4 for details.



Migrating from 3.2 to 3.3

This chapter contains the instructions for moving from VR-Forces 3.2 to VR-Forces 3.3.

3.1. Changes to Entity and Object Naming

In previous versions of VR-Forces, an entity's name and echelon ID were essentially the same thing. In this release, an entity's name and echelon ID are separate. The principle effects in the GUI of this change are:

- Entity names are a distinct concept from echelon IDs
- Users can assign meaningful names to objects as long as the names are unique
- VR-Forces manages echelon IDs. Users cannot directly change them.
- You can display names, echelon IDs, or both in the GUI
- When you create a plan, the plan is still associated with an echelon ID, however all the tasks in the plan they require identification of another entity, such as Follow Entity, use the entity name. This means that you can create plans that task entities to interact with non-VR-Forces entities, because non-VR-Forces entities have names, but do not have echelon IDs.

VR-Forces assigns default names to new entities based on their category in the object parameters database, for example, USAtrRW 1, USAtrRW 2, and so on. Objects are named based on their object type, for example, Point 1, Point 2, and so on. You can change the name given to an entity or an object.

A name must be unique. Names have the following length restrictions (because VR-Forces uses the marking text to send the object names over the network):

- Entity - 11 characters
- Aggregates - 10 characters in HLA exercises, 32 characters for DIS.
- Control objects - 255 characters.

VR-Forces assigns each entity an echelon ID based on its position in the force hierarchy. You cannot change an entity's echelon ID. An entity's echelon ID changes if you reorganize it into or out of an aggregate. It can change if you turn on auto-reorganization and its aggregate unit reorganizes.

You can also set an entity's label. Labels are arbitrary strings that you can assign to objects. They do not have to be unique.

3.1.1. Backwards Compatibility of Object Names

When you open a scenario created with a previous version of VR-Forces, VR-Forces will assign a name (based on the site ID and application number) to any entity whose echelon ID string exceeds the allowable number of characters for the entity type under the new naming scheme. If the echelon ID is less than the length allowed for the entity's name, the echelon ID is assigned as the name. For all practical purposes, this means that most individual entities, and HLA aggregates will be renamed.

All entities will retain their previous echelon IDs.

Plans will be updated to replace echelon IDs with names, where necessary. We recommend that after you open a legacy scenario, that you save the scenario and reload it to synchronize plans and entity names.

3.2. Backward Compatibility of Object Parameters Database Files

If you created a custom object parameters database under a previous version of VR-Forces, you must edit it to use it with VR-Forces 3.3. We recommend you make a backup copy of the parameter database before making any changes. The following changes affect object parameters database files:

- ♦ Subcomponent lists no longer include an entry for the Organization Manager. You should delete these entries.
- ♦ The M1A2 entry has a machine gun as a multiple-weapon example only.
- ♦ Kegler missiles (kegler-missile-platform entry) have the object type: 1 (2 4 222 1 3 -1 -1).
- ♦ The M1A2 has an anti-radar damage entry. Its damage model specification now includes an entry for munition type 2 4 222 1 -1 -1 -1.
- ♦ The damage model specification for the generic CIS attack helicopter platform has a Maverick (munition type 2 2 225 1 4 -1 -1) entry.
- ♦ Added a new civilian vehicle entry (object type 1 (1 1 -1 27 -1 -1 -1)).
- ♦ All of the radio string types specified under entities' local-objects lists have changed as follows:
 - "local-radio" is replaced by "channel-radio"
 - "local-command-radio" is replaced by "command-channel-radio"
- ♦ The radio entry has been removed from all remote-objects lists (radios are no longer created for remote entities).
- ♦ The category string parameter has been added to the specification for control object entries. These strings are used to generate more intuitive object names for these objects (e.g. "Route 2"). The new category strings are as follows:
 - Point
 - Line
 - Route
 - Area
- ♦ If you have added or modified a fixed-wing entries, change your entry to reflect the above changes described in "Updated Fixed Wing Model" on page 9 of VR-Forces 3.3-ngc Release Notes.
- ♦ All entries that specified a component type of "automotive-actuator" in their descriptor list, have been changed from a component-descriptor-type of "actuator-component-descriptor" to the more specific type "automotive-component-descriptor". If you have added any new entries that specify an "automotive-actuator", make sure you change the component-descriptor type to the new "automotive-component-descriptor" type.

3.3. Simulation API Changes

While we strove to preserve backward compatibility with previous 3.x versions, there are a number of API changes that had to be made. These were necessary to support checkpointing, multiple weapon systems, a revised fixed-wing model, a new echelon organization scheme, and a revised radio model.

Many of the API changes in this release had to be made to support checkpointing. To more completely capture an entity's state as part of a scenario save, we had to make more of the 'process state' maintained by an entity's *DtSimComponents* part of the data saved to the order of battle file. Because the contents of an entity's *DtSimComponents* are not saved in a scenario, we had to move those state member variables out of the *DtSimComponent* classes and into new *DtVrfProcessStateRepository* classes, which are saved as part of an entity's state. The new process state repository classes are designed to hold the state data associated with one or more component models of a given process (for example, movement, execution of a weapon system). If you were directly using a member variable that has moved (for example, out of a *DtSimComponent* and into a corresponding process state repository), then you will have to modify your code. Instead of accessing the old member variable in a *DtSimComponent* class directly, you will have to replace this code with calls to the accessors and mutators in the corresponding process state repository class.

The interface to the *DtSimComponent* constructor has changed. The constructor now takes a pointer to the entity it belongs to and a pointer to the component descriptor used to specify a component. This change affects the component factory and all derived component classes. The *setEntity()* and *setComponentDescriptor()* member functions are obsolete. They have been retained in the base class and are still called in the Component Manager for backward compatibility. Any functionality previously done in *setEntity()* has been moved to the constructor or the *init()* member as appropriate. The following related changes have been made:

- ♦ The acquire-time parameter has been removed from *DtVrfObjectParameters* (*vrfObjParams.h*).
- ♦ In *DtSimComponent* (*simCmpnt.h*), we added the ability for a component to connect to multiple port interfaces, even if they all have the same name. The *DtTargetDetectionSensor* has been modified to support connecting to multiple port interfaces. This allows this sensor to feed the same target list to as many controllers as may want it.
- ♦ In *DtVrfObjectStateRepository* (*vrfObjSR.h*) we removed the current target, target acquired, and user target override members. These parameters have migrated to the weapon PSR used by each weapon subsystem. This enables multiple weapons per platform. This affects the following class interfaces:
 - *balGun.h*
 - *balGunCtrl.h*
 - *baseWeapCtrl.h*
 - *indFireCtrl.h*
 - *indLifeFormAcqr.h*
 - *launchCtrl.h*
 - *turretAcqr.h*
 - *turretAct.h*
 - *turretCntlr.h*
 - *turretScan.h*
 - *weaponAct.h*
- ♦ *DtSimComponent* (*simCmpnt.h*) has a *postAddComponentsInit()* member function that gets called after all of an entity's components have been created and their *init()* functions called. It provides a convenient place to do any additional initialization of a component that is dependent on the existence of any other components. The implementation of *postAddComponentsInit()* in the *DtSimComponent* base class is empty. Derived classes can override this function to do any processing that must take place after all components have been created. In particular, we use it to do the lookups of process state repositories for those components that need to obtain pointers to a process state repositories. It can only be done successfully after the component responsible for creating the process state repository has been created, so process state repository lookups are performed there.

- ♦ Formation type has moved out of *DtSimSubordinateManager* (*subordMgr.h*) and into the new *DtPseudoAggregateMovementPSR* (*pseudoMovePSR.h*), so it can be saved as part of scenario.
- ♦ The *myActuatorParametersFound* and *getAcutatorParameters()* members of *DtFixedWingPilotController* (*fixWngPltCtl.h*) and *DtRotaryWingPilotController* (*rotWngPltCtl.h*) have been removed from these classes. The concept of a process state repository providing a shared container for process state (shared amongst components) renders these members obsolete. For example, we no longer need to have the *DtFixedWingPilotController* look up the *DtFixedWingActuator* to find out about flight-related parameters. Instead, both components can set/get these parameters in the *DtFixedWingFlightPSR*.

3.3.1. Changes to Weapons Systems

The new *DtVrfWeaponPSR* is a process state repository for sharing data (and reading/writing) data needed by any weapon system. This PSR replaces the entity SR current target and target acquired fields, as well as encapsulating the ammo loading system that was spread among the different weapon controllers. Because an entity can have multiple, uniquely named process state repositories, we can now support multiple weapon systems on a single platform. We were previously limited by the fact that there was only one set of current target and target acquired fields in the entity state repository. This had meant only one weapon system would work.

We have added the *DtMunitionLoader* (*munLoader.h*) class, which encapsulates ammunition loading code that was formerly distributed among various controllers. The *DtWeaponProcessSR* has a munition loader.

Example of Configuring Multiple Weapons on a Platform

We have added a machine gun to the M1A2 entry as an example of configuring an entity with multiple weapon systems (an M1A2 does not have a M16 mounted on it). We extended the target detection files (M1A2-fgt.det, M1A2-ins.det, M1A2-ots.det) to include probability of detection associated with lifeform entities (because the M1A2 example now includes a machine gun).

3.3.2. Other Changes Related to Weapons Systems

- ♦ In the simple rotary wing model, the pilot controller attempts to point the RW at the current target. Since platforms may now have multiple targets, this will attempt to align itself to the target acquired by the first weapon if found.
- ♦ The Local Entity Set Controller has been modified to provide minimal support for multiple weapons per platform. Since there is no mechanism to indicate which weapon should get the target specified, it is given to the first weapon system found.
- ♦ Fire-now behavior for entities with multiple ballistic guns handles requests to fire an entity's weapon by choosing the first weapon it finds on its platform.
- ♦ It is not realistic to set a lifeform's posture to weapon-in-fire position through the set data request mechanism. This posture should only be used when the entity fires its weapon. When a back-end receives a set data request specifying the weapon-in-fire posture, it maps that posture to the nearest reasonable posture, weapon-deployed.
- ♦ We were using world location instead of entity location when handling detonation interactions of type *DtDetResEntityProximate*.

3.3.3. Changes to the Organization Manager

The old organization manager class (*DtOrganizationManager*) that was a component of each entity has been removed.

A new class (also called *DtOrganizationManager*) is a member of *DtVrfObjectManager* has an API to view the current organization hierarchy, as well as to make any modifications to the hierarchy. See the *VR-Forces Developer's Guide* for more information of using the new organization manager. All calls to change the organization should go through this new organization manager.

Each organized entity now also has an Organization Controller component. On entities, this component is responsible only for keeping the echelon ID up to date. On pseudo-aggregates, this component sets and tracks echelon IDs of its subordinates, as the v3.2 organization controller did.

DtVrfObjectBaseStateRepository now has an accessor and mutator for the object's echelon ID.

The *DtRadioMessage* class has new members for channel number and flag telling whether the recipient is an entity name, or an entity echelon ID. This flag defaults to entity name for backwards compatibility. Call `setReceiverIsEchelonId(DtTrue)` to address a message to an echelon ID.

3.3.4. Changes to Radios

The simulation radios used for entities to communicate with each other have been changed. The old *DtSimLocalRadio* class has been replaced with *DtChannelRadio* and *DtSimLocalCommandRadio* has been replaced with *DtCommandChannelRadio*. Remote radios have been removed. These new radios inherit from the same *DtSimBaseRadio* as the old radios did, and work in much the same way as the old radios from the API perspective.

All the radio receiver and radio transmitter classes have been removed, and there is a *DtRadioMessageExecutive* is used directly in the radio to simplify customization of the radios.

Radio Net class (*DtSimRadioNet*) has been removed. The new channel radios send and receive messages addressed to the channel number they are tuned to. Everyone else tuned to the same channel receives the messages, conceptually the same as the old radio nets.

The following classes have been removed in VR-Forces 3.3:

- ♦ *DtSimRadioNetsimRadioNet.h*
- ♦ *DtSimRadioReceiverradioRcvr.h*
- ♦ *DtSimRadioTransmitterradioXmitter.h*
- ♦ *DtReceiverTransmitterrcvrTrans.h*
- ♦ *DtCommandRadioTransmittercmdXmitter.h*
- ♦ *DtSimLocalRadiolclRadio.h*
- ♦ *DtLocalRadioReceiverlclRadRcvr.h*
- ♦ *DtLocalReceiverTransmitterlclRcvrTrans.h*
- ♦ *DtUnitRadioTransmitterunitXmitter.h*
- ♦ *DtLocalUnitReceiverTransmitterlclUnitRT.h*
- ♦ *DtSimLocalCommandRadiolclCmdRadio.h*
- ♦ *DtLocalCommandReceiverTransmitterlclCmdRT.h*
- ♦ *DtSimRemoteRadioremRadio.h*
- ♦ *DtRemoteRadioReceiverremRadRcvr.h*
- ♦ *DtRemoteReceiverTransmitterremRcvrTrans.h*
- ♦ *DtRemoteCommandReceiverTransmitterremCmdRT.h*
- ♦ *DtSimRemoteCommandRadioremCmdRadio.h*
- ♦ *DtRemoteUnitReceiverTransmitterremUnitRT.h*

3.3.5. Miscellaneous Simulation API Changes

The fixed-wing model now automatically sets its ordered altitude to the altitude of the waypoint, current route vertex, or entity to follow when executing movement tasks. This results in the fixed-wing entity immediately attempting to move to the desired altitude.

3.4. Changes to the Remote Control API

The Remote Control API has changed as follows:

- New *DtBackend* class (*backend.h*)
- *DtVrfBackendListener* (*vrfBeListener.h*) - Added a member function to return the number of back-ends currently loaded in a scenario.
- Changes to *vrfController.h* - Added an accessor for *DtVrfBackendListener*. Added a member function to return the number of back-ends currently loaded in a scenario. You can use the *DtBackend* object to see which back-ends have a scenario loaded by inspecting its *isLoading()* member function.

3.5. Changes to the GUI API

The GUI API has the following changes:

- *myTMMapArea* no longer exists as a member variable of the *DtTMWindow* class. It must be replaced with a function call to *areaMap()*. Any code that references this member variable will no longer compile
- The *DtTMWindow* class has been changed to be a subclass of *DtBaseWindow*, so, any creator functions that look like this:

```
static DtTMWindow * create(QWidget* parent, const char* name, WFlags
    flags);
```

must be changed to look like this:

```
static DtBaseWindow * create(QWidget* parent, const char* name, WFlags
    flags);
```

to have the code compile properly.

- The *initMapArea* of the *DtPvdWindow* class has changed its prototype from:

```
virtual bool initMapArea(const DtTMFactory* f);
```

to:

```
virtual bool initMapArea(DtTMMapArea* area, const DtTMFactory* f);
```

This accounts for the fact that a window can now support multiple map areas.

- The *initMiniMap* of the *DtPvdWindow* class has changed its prototype from:

```
virtual bool initMiniMap(const DtTMFactory* f);
```

to:

```
virtual bool initMiniMap(const DtTMFactory* f, QWidget* parent = 0);
```

3.6. Porting MÄK Plan View Display Examples to VR-Forces

Some of the examples supplied with VR-Forces to illustrate use of the GUI API were written for the MÄK Plan View Display (PVD) API. The PVD examples are contained in a workspace called `pvdExamples.dsw`. If you build these examples as they are supplied, they will only allow you to open databases and view entities that are part of an existing simulation. In other words, they just provide PVD features.

The examples that are considered to be PVD examples:

- ♦ `drawMissileImpact`
- ♦ `unitStatus`
- ♦ `viewControlObjects`
- ♦ `windowsInfoExtension`
- ♦ `xMarksTheSpot`.

You can edit the PVD examples to provide full VR-Forces features, as follows:

1. Modify `main.cxx` to reference *`DtVrfGuiApplication`* and *`DtVrfGuiFactory`* rather than *`DtPvdApplication`* and *`DtPvdFactory`* as follows:

```
#include "vrfGuiFactory.h" // was--> #include "pvdFactory.h"
#include "vrfGuiApplication.h" // was --> #include "pvdApplication.h"
#include "MachineTypes.h"
#include "userWindow.h"
#include "userMapArea.h"
#include "userSymbolMapper.h"
int main(int argc, char* argv[])
{
    DtVrfGuiFactory pf; // was--> DtPvdFactory pf;
    DtVrfGuiApplication pa; // was --> DtPvdApplication pa(argc, argv);
    pf.setMainWindowCreatorFcn(MyPvdUserWindow::create);
    if (!pa.init(&pf))
    {
        return -1;
    }
    pa.fileNewWindow();
    return pa.exec();
}
```

2. Each file that contains an instance of a subclass of a PVD object (for example, *`DtPvdMapArea`*), that must also be updated to reference the VRF version of the object. For example (from `xMarksTheSpot` *`userMapArea.h`*):

```
#include "vrfGuiMapArea.h" // was --> #include "pvdMapArea.h"
class MyPvdUserMapArea : public DtVrfGuiMapArea // was --> class
    MyPvdUserMapArea : public DtPvdMapArea
```

3. In the `.cxx` file that also contains the object being updated, you must change any superclass reference from the `pvdObject` to the `vrfGui` representation. For example:

```
void MyPvdUserMapArea::startIntervisibilityCreation(int iMode)
{
```

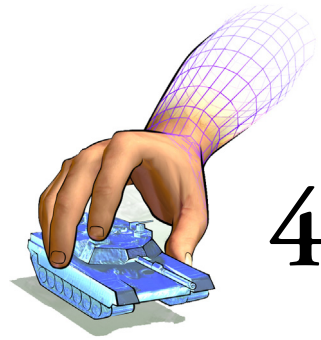
```
stopObjectCreation();  
DtVrfGuiMapArea::startIntervisibilityCreation(iMode); // was -->  
    DtPvdMapArea::startIntervisibilityCreation(iMode);  
}
```

You do not have to change the class object name from PVD (although you are not required to keep it the same name) in order to modify the application for VR-Forces usage.

4. Repeat this process for all objects that are a subclass of a *DtPvd** object. Objects that are not a subclass of *DtPvd* (such as a the *userDrawer* class in the *xMarks-TheSpot* example) do not need to be modified.



You must link with the VR-Forces libraries. You can copy the link lines from any VR-Forces example .dsp.



Migrating from 3.3 to 3.4

This chapter contains the instructions for moving from VR-Forces 3.3 to VR-Forces 3.4.

4.1. Updated GUI API

The updated GUI API is documented in *VR-Forces Developer's Guide*. For directions for porting your existing GUI code to the new version, see [“GUI API Migration Instructions,”](#) on page 4-4. The GUI API has the following new examples:

- ♦ filterMenu – This example will show you how to add new menus and menu items to an existing application and hook up those menu items to provide functionality for your new application.
- ♦ trackHistory – This example shows you how to use the *DtMtlSymbolMapper* and *DtPvdSymbolUpdater* to add new symbols to the application and then modify them with new data.
- ♦ autoRewind – This example shows you how to install and use an application level event controller and event handler.

4.2. New Formats for Scenario Files

As a result of changes made to plan files and other changes in this release, scenarios created with previous versions of VR-Forces may have problems running in VR-Forces 3.4. When you open a scenario created with a previous version of VR-Forces, you are prompted to upgrade the scenario to the new format. If you click Yes, a Save As dialog box opens. Save the scenario. (You may want to backup old scenarios before you upgrade to the new format.)

4.3. Terrain API Examples

VR-Forces has three new examples to support the Terrain API:

- ♦ createTerrainDb - shows how to create a simple terrain database using the terrain API.
- ♦ modifyTerrainDb - shows how to modify an existing terrain database (deform the terrain skin) using the terrain API.
- ♦ addTerrainFeatures - shows how to add terrain features (vector data) to a terrain database using the terrain API.

VR-Forces includes the following new terrain databases to support the examples:

- ♦ simpleTerrain.gdb - Simple flat (featureless) terrain generated by the createTerrain example.
- ♦ simpleFeatures.gdb - Generated by the addFeatures example. It is a modified version of simpleTerrain.gdb (added a few features to it (trees, building, river).
- ♦ modifiedSimpleTerrain.gdb - Generated by the modifyTerrain example. It is a modified version of simpleTerrain.gdb (terrain skin has been altered to have a 60 meter deep depression in one corner).

4.4. Changes to Network Representation of DtSimTasks

The *DtSimTask* class (base class for all tasks) has changed its implementation of `netRepSize()`, `setToNet()`, and `setFromNet()`. This is because the base *DtSimTask* class now reads and writes data to the network representation of the task. If you have created your own derived kind of *DtSimTask*, you must call down to the base class version of these functions in your derived class. See the `addTaskSim` or `addTaskGui` examples for an example of implementing these functions in a derived kind of *DtSimTask*.

In your implementation of `netRepSize()`, you must down to the base class `netRepSize()`, and add that to the size of the network representation for the task.

```
int DtRetreatTask::netRepSize() const
{
    int size;
    size = DtSimTask::netRepSize();
    size += sizeof(DtNetRetreatTask);
    size += myRetreatKind.length();
    size = (int) ceil(size / 8.) * 8;
    return size;
}
```

In your implementation of `setFromNet()`, you must call down to the base class version of `setFromNet()`, to ensure that base class data is retrieved from the network representation. Use the base class `netRepSize()` to find the correct offset into the network representation to start the derived class's data.

```
DtBoolean DtRetreatTask::setFromNet(const char* contentBuffer,
unsigned contentSize)
{
    if(!DtSimTask::setFromNet(contentBuffer,contentSize))
    {
        return DtFalse;
    }
    DtNetRetreatTask* taskBuffer;
    DtU16 stringLen;
    char buf[256];
    const char* stringPtr;
    const char* bufferOffset = contentBuffer + DtSimTask::netRepSize();
    taskBuffer = (DtNetRetreatTask*)bufferOffset;
    . . .
}
```

Similarly, in `setToNet()`, you must call down to the base class version of `setToNet()`, to ensure that the base class data is copied to the network representation. Use the base class `netRepSize()` to find the correct offset into the network to copy the derived class data to.

```
DtBoolean DtRetreatTask::setToNet()
{
    if(!DtSimTask::setToNet())
    {
        return DtFalse;
    }
    if (!netRep())
    {
        return DtFalse;
    }
}
```

```
char* bufferOffset = myNetRep + DtSimTask::netRepSize();
DtNetRetreatTask* taskBuffer = (DtNetRetreatTask*)bufferOffset;
. . .
```

4.5. Miscellaneous API Changes

VR-Forces 3.4 includes the following additional changes to the APIs:

- Added MIN_USER_RADIO_MESSAGE_TYPE #define to *radioMsgTypes.h*. This value represents the lowest (starting) value for user-defined *DtSimTask* types. Toolkit users should be aware of this value and choose integer types for their derived *DtSimTask* classes greater than or equal to this value.
- Added MIN_USER_INTERFACE_MESSAGE_TYPE and MAX_USER_INTERFACE_MESSAGE_TYPE to *msgTypes.h* to define a safe range of integer types toolkit users can use for their own custom interface messages (derived *DtSimInterfaceContent* types).
- Allow plans to be assigned by name, to overcome bug in which echelon ID is not always valid at the time plan is assigned.
- The simplePlan and simpleCGF examples are now console windows, reducing possibility of losing track of 'windowless' copies of the process.
- The objectManager example shows how to override a high-level manager.
- The target detection sensor tests for intersections with the vector network. Therefore, if your database has vector data, the sensor will be blocked by features such as buildings, trees, and so on.
- VR-Forces is now based on standard C++ IO streams.

4.6. GUI API Migration Instructions

The VR-Forces GUI API has been rewritten to organize its functionality into smaller class units and make the subclassing and adding of functionality easier. *VR-Forces Developer's Guide* fully describes the updated API. The following sections summarize the changes made and provide migration instructions for applications written with previous versions of the API. The examples have been modified to work with the updated API, and where appropriate, the old code has been commented out and new code added to show how to migrate from the old API to the new.

4.6.1. Removed Classes

The *DtVrfGuiMapArea* class has been removed. Any class that you have derived from *DtVrfGuiMapArea* must now be changed to subclass *DtPvdMapArea*.

4.6.2. New Classes

Most of the functionality that used to exist in the *DtVrfGuiMapArea* and *DtVrfGuiWindow* objects has been divided among the following classes:

- ♦ *DtVrfGuiEventController*
- ♦ *DtVrfGuiAppEventController*
- ♦ Various menu object (*DtVrfGuiFileMenu*, for example) that encapsulate the creation of menus and menu items
- ♦ *DtVrfGuiRightClickMenu*
- ♦ *DtPvdMiddleClickMenu*.

These classes are explained in detail in *VR-Forces Developer's Guide*.

4.6.3. The Names of Factory Classes have Changed

The names of the default factories have changed to include the word “Default” in them. If you used to reference *DtVrfGuiFactory*, you must now reference *DtVrfGuiDefaultFactory*. Similarly, if you referenced *DtPvdFactory* you must now reference *DtPvdDefaultFactory*.

4.6.4. Changes to Existing Classes

The *DtMtlSymbolMapper* and *DtPvdSymbolUpdater* classes have been changed to allow for easier subclassing and modification of symbols.

The *DtMtlSymbolMapper* Class

DtMtlSymbolMapper has been reorganized from large member functions that create entire symbols into individual parts that allow you to override a very specific piece of functionality that you might want to change. Refer to *mtlSymbolMapper.h* for more information about the structure of the class.

How to Change Code that Subclasses from *DtMtlSymbolMapper*

If you override the `createSymbol()` member function, then you do not need to make any changes. If you override member functions such as `getEntitySymbolFromData()`, `getAggregateSymbolFromData()`, and so on, those member functions have changed to the following:

```
virtual DtSymbol* processCreateSymbol(const DtEntityModelData& data,
    void *usr);
virtual DtSymbol* processCreateSymbol(const DtAggregateModelData& data,
    void *usr);
virtual DtSymbol* processCreateSymbol(const DtControlObjectModelData&
    data, void *usr);
virtual DtSymbol* processCreateSymbol(const DtFireInterModelData& data,
    void *usr);
virtual DtSymbol* processCreateSymbol(const DtDetonateInterModelData&
    data, void *usr);
```

By overriding the new version of the member function, you should be able to use your code exactly as it exists today in the new member function.

The *DtPvdSymbolUpdater* Class

Like *DtMtlSymbolMapper*, *DtPvdSymbolUpdater* has been reorganized to provide more member functions that target specific functionality. Please refer to *pvdSymbolUpdater.h* for more information about the structure of the class.

How to Modify Code that Subclasses from *DtPvdSymbolUpdater*

If you currently subclass *DtPvdSymbolUpdater*, look at the header file. If you have all your code in the `updateEntitySymbol()` member function, there may be new member functions that are more appropriate for you to override (for example, in one of the specific update member functions). The general update member functions now have the following prototypes:

```
// Update entity attributes
virtual void updateSymbol(DtPvdEntitySymbol* es,
    const DtEntityModelData& d, void* usr) const;
// Update aggregate attributes
virtual void updateSymbol(DtPvdEntitySymbol* es,
    const DtAggregateModelData& d, void* usr) const;
// Update control object attributes
virtual void updateSymbol(DtNameLabelSymbol* es,
    const DtControlObjectModelData& d, void* usr) const;
// Note that this updateOptions is used by both entity and aggregate
// symbols
virtual void updateOptions(DtPvdEntitySymbol* es,
    const DtPvdDisplayOptions* opt, void* usr) const;
// Update options for name label symbols (waypoints, areas, etc.)
virtual void updateOptions(DtNameLabelSymbol* nls,
    const DtPvdDisplayOptions* opt, void* usr) const;
// Update options for interactions symbols (fire, detonate)
virtual void updateOptions(DtInterSymbol* nls,
    const DtPvdDisplayOptions* opt, void* usr) const;
```

You can override the member function that meets your need (*DtNameLabelSymbols* now represent control objects), calling the parent implementation first.

i

DtModelData and its subclasses now return the type of data that they represent. If you have a subclass of *DtModelData* that is a new type, you can assign the particular type of that model data by using the `DtNewModelDataType` macro to create a new model data type. You can then use this type to register creation and update functions in the *DtMtlSymbolCreator* and *DtPvdSymbolUpdater* classes.

4.6.5. Changes to the Menu System

The menu system has been removed from various `init()` functions in the window classes (*DtVrfGuiWindow*.) and placed into discrete objects that are subclassed from *DtMenuWidget* and *DtPopupMenu* (in *menuWidget.h* and *popupMenu.h*). These classes provide a simple interface to adding new menu items and action functions to menus. The header file *menuItems.h* describes the menus and menu items that are contained in the application and the base variables that you can use.

The *DtTaskActionGroup* and *DtSetActionGroup* Classes

DtTaskActionGroup and *DtSetAction* no longer have their creation functions set in the *DtVrfGuiFactory*. Their creation functions are now registered in the *DtMenuWidget* class.

Migrating Code that Subclasses from *DtTaskActionGroup* and *DtSetActionGroup*

To register your task and set menus to be created, you must do the following in *main.cxx*:

Old code:

```
// Tell the factory to create our instance of the task group
f.setTaskActionGroupCreatorFcn(MyTaskActionGroup::create);
```

New code:

```
#include "menuItems.h"
DtMenuWidget::globalReplacePopupMenu(DtTaskMenu,
    MyTaskActionGroup::create);
```

or:

```
DtMenuWidget::globalReplacePopupMenu(DtSetMenu,
    MySetActionGroup::create);
```

Change the `create()` prototype and object constructors to follow the new prototype:

Old code:

```
MyTaskActionGroup(QObject* parent, const char* name = 0,
    bool exclusive = TRUE);
static DtTaskActionGroup* create(QWidget* p, const char *name,
    bool enabled);
```

New code:

```
MyTaskActionGroup(QObject* parent, const char* name = 0);
static DtPopupMenu* create(QObject* parent, const char *name);
```

Replacing the *myMapArea* Member Variable

- ♦ The *myMapArea* member variable has been removed and can be replaced with a reference to the member function `mapArea()`.

Migrating Code that Adds Menu Items.

The `init()` function no longer exists and has been changed to a function called `initMenuItems()`.

Any menu items that you have created have to be created in a different form. Menu items are no longer created as `QActions`. They are now scheduled to be created in the `initMenuItems()` member function and then created and added to the menu in the `createMenuItems()` member function. Using the `addTaskGui` as an example, if your old code looks like this:

```
bool MyTaskActionGroup::init(DtPvdMapArea* mapArea)
{
    if(!DtTaskActionGroup::init(mapArea))
    {
        return false;
    }

    myTaskRetreatAction = new QAction( this, "myTaskRetreatAction" );
    myTaskRetreatAction->setText( trUtf8( "Retreat..." ) );
    myTaskRetreatAction->setMenuText( trUtf8( "Retreat..." ) );
    myTaskRetreatAction->setToolTip( trUtf8(
        "Task selected entity/aggregate to retreat." ) );

    // Connect the actions.
    connect
    (
        myTaskRetreatAction,
        SIGNAL(activated()),
        this,
        SLOT(taskSelectionRetreatTask())
    );

    return true;
}
```

You must change it to look like this:

```
bool MyTaskActionGroup::initMenuItems()
{
    if(!DtTaskActionGroup::initMenuItems())
    {
        return false;
    }

    insertMenuItem(DtNoMenuItem, DtUserMenuItemStart,
        trUtf8( "Retreat..." ), trUtf8( "Retreat..." ),
        trUtf8( "Task selected entity/aggregate to retreat." ));

    addActionAndValidatorForType(DtUserMenuItemStart, this,
        SLOT(taskSelectionRetreatTask()));

    return true;
}
```

Notes

- ♦ The changes that need to be made to subclasses of *DtTaskActionGroup* must also be made to subclasses of *DtSetActionGroup*. Use the description of changing *DtTaskActionGroup* subclasses as a template for changing *DtSetActionGroup* subclasses. If you look at the prototype of the `addActionAndValidatorForType()` member function, you will see that you can add a validator after the action in the final two parameters.
- ♦ The base `DtUserMenuItemStart` constant was used. If you define more than one type of menu item in your application, you must create different constant values starting at the `DtUserMenuItemStart` constant, using those as references in your `insertMenuItem()` and `addActionAndValidatorForType()` member functions. This will ensure that your new menu types do not conflict with any types defined by VR-Forces.

The application objects define the menu changes they want to make for the application (*DtVrfGuiApplication*), in the `initMenuSystems()` member function called from the `init()` member function. If you want to make modifications to the menu system, be sure to make them after the `init()` member function of the application object. You may also create your own subclass of the application object to create your menu system:

```
bool MyApplication::initMenuSystem()
{
    if (!DtVrfGuiApplication::initMenuSystem())
    {
        return false;
    }

    DtMenuWidget::globalReplacePopupMenu(DtViewMenu, MyViewMenu::create);
    DtMenuWidget::globalReplacePopupMenu(DtCreateMenu,
        MyCreateMenu::create);
    DtMenuWidget::globalReplacePopupMenu(DtEntitiesMenu,
        MyEntityMenu::create);
    DtMenuWidget::globalReplacePopupMenu(DtTaskMenu,
        MyTaskActionGroup::create);
    DtMenuWidget::globalReplacePopupMenu(DtOptionsMenu,
        MyOptionsMenu::create);

    return true;
}
```

See the `addTaskGui` and `mergedSet` examples for details of migrating code. See the `simpleScenarioPlayer` example for how to globally configure the main menu bar through the `main.cxx` file.

Adding Menu Items to Existing Menus

To add menu items to existing menus (for example the View menu), use one of the following methods.

- ♦ Use the window member function:

```
bool MyMainWindow::init(const DtTMFactory* factory, DtTMMiniMap* miniMap)
```

If you override this member function, you can add your new menu item to the View menu after calling the parent's `init()` member function.



This method will only allow you to append menu items to an existing menu.

The following code is from the `fogOfWar` example:

```
bool MyMainWindow::init(const DtTMFactory* factory, DtTMMiniMap* miniMap)
{
    if(!DtVrfGuiWindow::init(factory, miniMap))
    {
        return false;
    }

    // Now, add our new menu item -- uses new 3.4 toolkit changes
    DtPopupMenu* viewMenu = myMenuWidget->menuForType(DtViewMenu);

    viewMenu->addMenuItemSeparator(0);
    viewMenu->insertMenuItem(DtNoMenuItem, DtUserMenuItemStart,
        trUtf8( "Switch to Friendly" ), trUtf8( "Switch to Friendly" ),
        trUtf8(" Switch between showing all, friendly or opposing forces in
        fog of war filter"));
    viewMenu->addActionAndValidatorForType(DtUserMenuItemStart, this,
        SLOT(choiceSwitch()));

    return true;
}
```

- ♦ Override the menu.

If you want to create your own menu class, you must override the *DtVrfGuiViewMenu* class, install your subclass as the menu to be created, and then create the new menu item as follows:

```
// The main.cxx file contains global changes to the default
// DtVrfGuiApplication and factory objects.
// It will also allow for modification of the default menu system

#include "vrfGuiDefaultFactory.h"
#include "vrfGuiDefaultFactory.h"
#include "vrfGuiApplication.h"
#include "menuItems.h"
#include "menuWidget.h"
#include "myViewMenu.h"

int main(int argc, char* argv[])
{
    // Set our aggregate filter handler to create a new fog of war filter.
    // This must be done first as when the app is inited it will use the
    // default handler

    DtEventController::addHandler(DtEchelonViewHandlerType,
        MyFogOfWarHandler::create);

    DtVrfGuiDefaultFactory factory;
    DtVrfGuiApplication app(argc, argv);
```

```

    if (!app.init(&factory))
    {
        return -1;
    }

    // Replace the default view menu with our new view menu. This member
    // function must be after the applications init() member function such
    // that our definition is not overwritten.
    DtMenuWidget::globalReplacePopupMenu(DtViewMenu, MyViewMenu::create);

    app.fileNewWindow();

    return app.exec();
}

#ifndef MYVIEWMENU_H
#define MYVIEWMENU_H

#include "vrfGuiViewMenu.h"

class MyViewMenu : public DtVrfGuiViewMenu
{
    Q_OBJECT

public:
    MyViewMenu ( QObject * parent, const char * name = 0 );
    virtual ~MyViewMenu ();

    // Pure virtual overrides
protected:
    // Initializes and maps the following menu items. Note these items are
    // inserted as toggle menu items
    //
    // DtUserMenuItemStart to myOwner->choiceSwitch()
    // DtVrfGuiViewMenu items (see vrfGuiViewMenu.h)
    virtual bool initMenuItems();

public:
    // Creator function that will return an instance of this object
    static DtPopupMenu* create(QObject* parent, const char* name = NULL);
};

#endif

#include "myViewMenu.h"
#include "menuItems.h"

MyViewMenu::MyViewMenu(QObject* parent, const char* name) :
DtVrfGuiViewMenu(parent, name)
{
}

// Initialize our new version of the view menu by creating the stock view
// menu first and then adding our new view item which will switch the
// force type of the fog of war filter
bool MyViewMenu::initMenuItems()
{
    if (!DtVrfGuiViewMenu::initMenuItems())
    {
        return false;
    }
}

```

```

    }

    addMenuItemSeparator(0);

    insertMenuItem(DtNoMenuItem, DtUserMenuItemStart, trUtf8( "Switch to
    Friendly" ),
        trUtf8( "Switch to Friendly" ), trUtf8(" Switch between showing all,
        friendly or opposing forces in fog of war filter"));
    addActionAndValidatorForType(DtUserMenuItemStart, myOwner,
        SLOT(choiceSwitch()));}
}

DtPopupMenu* MyViewMenu::create(QObject* parent, const char* name)
{
    return new MyViewMenu(parent, name);
}

```

Refer to the fogOfWar example for an example of how to modify an existing menu from the init() member function. Refer to the filterMenu example for an example of how to create and install a new popup menu.

Validators for Menu Items

The validator map has been replaced with a function that uses Qt signals and slots to call validation functions for menu items. The addActionAndValidatorForType() member function allows you to assign actions and validator member functions for the menu items being created. Actions are called when a user selects a menu item. Validators are called each time the user displays a popup menu.

(From *popupMenu.h*)

```

// Adds a new action/validator pair. Supply NULL to have no action. The
// items passed are SLOTS that will be called via a connection to
// either the activated signal from the QAction (for the action) or
// the validateItems signal if something needs to be validated. The
// QObject* supplied is the item that will be signalled in response to
// both action and validator
virtual void addActionAndValidatorForType(int iType, QObject* aCall,
    const char* action, QObject* vCall = NULL,
    const char* validator = NULL);

#include "menuItems.h"

// Define our new menu item constant so we do not conflict with product
// menu items
const int DtMyMenuItem = DtUserMenuItemStart;

bool DtMyMenu::initMenuItems()
{
    // Call parent class such that our new menu item is inserted after all
    // other
    if (!DtVrfGuiViewMenu::initMenuItems())
    {
        return false;
    }

    // Now add our new item and validation

```

```

insertMenuItem(DtNoMenuItem, DtMyMenuItem, trUtf8("My Action"),
               trUtf8("My Action"));
addActionAndValidatorForType(DtMyMenuItem, this, SLOT(myAction()),
                             this, SLOT(myValidator()))

return true;
}

// Enable menu item only if a terrain database is loaded. Called when the
// popup menu is displayed or activated
void DtMyMenu::myValidator()
{
    // Get a handle to our menu action
    QAction* action = menuItemForType(DtMyMenuItem);

    if (DtTMAApplication::app()->terrainDatabase())
    {
        action->setEnabled(true);
    }
    else
    {
        action->setEnabled(false);
    }
}

```

See the `unitStatus` example for an example of creating a view menu subclass.

The View Menu

If you have created custom toolbars that you added to the View menu, and you toggle those menu items on and off, the way that you add the toolbars to the View menu has changed. To convert to the new system, you must do the following:

1. Subclass the `DtVrfGuiViewMenu` as your own class:

```

#ifndef DtMyViewMenu_H
#define DtMyViewMenu_H

// VRF includes
#include "vrfGuiViewMenu.h"

class DtMyViewMenu : public DtVrfGuiViewMenu
{
    Q_OBJECT

public:
    DtMyViewMenu ( QObject * parent, const char * name = 0 );
    virtual ~DtMyViewMenu ();

    // Initializes and maps the following menu items
    virtual bool initMenuItems();

public:
    static DtPopupMenu* create(QObject* parent, const char* name = NULL);
};

```

```
#endif
```

1. Create an object that adds your the options to the View menu. If you have more than one option, start at *DtUserMenuItemStart* and increase from there:

```
#include <myViewMenu.h>
#include <menuItems.h>

DtNewMenuItemType(DtMySwitchItem, 1);

DtMyViewMenu::DtMyViewMenu( QObject * parent, const char * name):
    DtVrfGuiViewMenu( parent, name )
{
}

DtMyViewMenu::~DtMyViewMenu()
{
}

bool DtMyViewMenu::initMenuItems()
{
    // By calling the view menu parent method last, this ensures our menu
    // item will be shown first
    insertToggleMenuItem(DtNoMenuItem, DtMySwitchItem, trUtf8( "Switch
Toolbar" ),
        trUtf8( "Switch Toolbar" ), trUtf8( "Activates my toolbar." ));

    // Unless you want to do some custom handling, all toolbars are now
    // turned on/off via the toggleToolbar() member function.
    // The toggleToolbar() function is defined in baseWindow.h and will use
    // the myViewMenuToolbarMap (see below) to match our menu item to a
    // specific toolbar
    addActionAndValidatorForType(DtMySwitchItem , myOwner,
        SLOT(toggleToolbar(bool)));

    if (!DtVrfGuiViewMenu::initMenuItems())
    {
        return false;
    }

    return true;
}

DtPopupMenu* DtMyViewMenu::create(QObject* parent, const char* name)
{
    return new DtMyViewMenu(parent, name);
}
```

1. Subclass *DtVrfGuiWindow* and override the *initViewMenuToolbarMap()* member function to map your new toolbar menu item to the toolbar it affects:

```
// Must do this mapping to be sure our toolbar can be shown/hidden
void DtMyUserWindow::initViewMenuToolbarMap()
{
    myViewMenuToolbarMap.insert(DtUserMenuItemStart , myNewToolBar);
    DtVrfGuiWindow::initViewMenuToolbarMap();
}
```

1. Install your new View menu in *main.cxx* after you create the application object:

```
#include "vrfGuiDefaultFactory.h"
```

```
#include "vrfGuiApplication.h"

#include "menuItems.h"
#include "menuWidget.h"
#include "myViewMenu.h"

int main(int argc, char* argv[])
{
    DtVrfGuiDefaultFactory f;
    DtVrfGuiApplication a(argc, argv);

    if (!a.init(&f))
    {
        return -1;
    }

    // Must be done after init so application init does
    // not override changes
    DtMenuWidget::globalReplacePopupMenu(DtViewMenu,
        DtMyViewMenu::create);

    a.fileNewWindow();

    return a.exec();
}
```

1. Remove any code in your window subclass that used to turn the toolbar on and off.

Right and Middle Click Menus

Right-click and middle-click menus are now their own objects and are set in the factory using `setCreateRightClickContextMenuCreatorFcn()` and `setCreateMiddleClickContextMenuCreatorFcn()`. The functions that used to be in the windows classes (for example, `updateRightClickSingleEntity()`) are now in the *DtVrfGuiRightClickMenu* class (see *vrfGuiRightClickMenu.h*). If you overrode these member functions to add your own right click functionality, you must now subclass *DtVrfGuiRightClickMenu* and reimplement the member functions that you implemented in the windows class. Look at `addMenuItem()` in *contextMenu.h*. This member function takes a menu item type you want to add (as defined in *menuItems.h* or your own menu item types), and optionally, a separator and where to place the item in the menu:

```
void DtVrfGuiRightClickMenu::updateRightClickSingleEntity()
{
    addMenu(tr("Task"), DtTaskMenu);
    addMenu(tr("Set"), DtSetMenu, true);
    addMenuForGroup(tr("Intervisibility"), DtIntervisibilityMenu,
        DtIntervisibilityAction, true);
    addMenuItem(DtEditAction, true);
    addMenuItem(DtEntitySkipTaskAction);
    addMenuItem(DtEntityEditPlanAction);
    addMenuItem(DtEntityViewPlanAction);
    addMenuItem(DtEntityAbandonPlanAction, true);
    addMenuItem(DtCollapseAggregateAction, true);
    addMenuItem(DtEntityInfoAction);
    addMenuItem(DtEntityListAction);
    addMenuItem(DtEchelonPanelAction, true);
    addMenuItem(DtCenterSelectionAction);
}
```

```

addItem(DtTrackSelectionAction);
addItem(DtClearTrackingAction, true);
addItem(DtEntityEnableJoystickAction);
addItem(DtEntityDisableJoystickAction, true);
addItem(DtStealthAttachAction);
addItem(DtStealthDetachAction);
addItem(DtStealthTeleportAction, true);
addItem(DtEntityDeleteAction);
}

```

4.6.6. Event Controllers and Event Handlers

Most of the functionality that was contained in the map areas, applications, and windows has been moved into event controllers (classes such as *DtVrfGuiEventController* and *DtVrfGuiAppEventController*) and event handlers (classes such as *DtCreateAreaHandler* and *DtBatchScenarioHandler*). An event controller is a manager of pieces of functionality (event handlers). It installs and runs events based upon actions taken in the interface. You can also install default handlers that are always present and perform some piece of functionality every tick, or when some keyboard or mouse event takes place (for example, the navigation handler or the drag and drop handler). *VR-Forces Developer's Guide* describes the different types of handlers and the event controllers that are in the system.

Accessing the Remote Controller

If you used to access the remote controller as:

```

DtVrfGuiApplication* app =
    dynamic_cast<DtVrfGuiApplication*>(DtTMApplication::app());
DtVrfRemoteController* controller = app->remoteController();

```

This has changed to:

```

DtVrfRemoteController* controller =
    DtVrfGuiAppEventController::remoteController();

```

Also, if you subclass the application object and reference the remote controller member variable, you can change that to reference the remote controller as above.

Creating Your Own Remote Controller

The remote controller has been moved to the *DtVrfGuiAppEventController*, and an accessor placed within that handler to set your version of the remote controller. If you used to create a version of the remote controller by subclassing the *setupRemoteController()* member function, that member function no longer works. After you initialize the application, you can reference the application controller as follows to set your new remote controller:

```

#include "vrfGuiDefaultFactory.h"
#include "vrfGuiApplication.h"
#include "vrfGuiAppEventController.h"

#include "myRemoteController.h"

```

```
int main(int argc, char* argv[])
{
    DtVrfGuiDefaultFactory f;

    DtVrfGuiApplication a(argc, argv);

    if (!a.init(&f))
    {
        return -1;
    }

    MyRemoteController* remoteControl = new MyRemoteController;

    remoteControl->init(DtVrfGuiAppEventController::appController()->
        messageInterface());
    DtVrfGuiAppEventController::appController()->
        setRemoteController(remoteControl);

    a.fileNewWindow();

    return a.exec();
}
```

You no longer need to subclass in order to set your remote controller.

4.6.7. The *DtEntityInfoDialog* Class

The Entity Information dialog box has been reorganized to allow for easier construction and code reuse by the Aggregate Information dialog box.

How to Modify Code That Subclasses from *DtEntityInfoDialog*

If you override the constructor to add new coordinate systems for the coordinate system list, you must now override and install a separate class in the factory. A new class called *DtBaseEntityInfoWidget* has been created to hold the coordinate system and common display information (algorithm, location, velocity, and so on) that is also used by the *DtAggregateInfoDialog* class.

You must subclass the *DtBaseEntityInfoWidget* class and install it as the widget to be created when an entity information or aggregate information window is created via the *setBaseEntityInfoWidgetCreatorFcn()* member function in the factory. You can then override the *init()* member function of the *DtBaseEntityInfoWidget* class, call the parent *init()*, and then remove your code from your subclass of the *DtEntityInfoDialog* class and place it in the *DtBaseEntityInfoWidget* *init()*. The various update member functions for updating velocity, location, and so on have also been moved to this new class. If your *DtEntityInfoDialog* implements these functions, then move your implementation to the *DtBaseEntityInfoWidget* class.

4.6.8. Editing Plan View Display Examples for Use with VR-Forces

VR-Forces includes examples that were written specifically for the MÄK Plan View Display, for example, the xMarksTheSpot example. When you compile and run the examples, they will look like the Plan View Display, not VR-Forces. To add full VR-Forces functionality to the examples, you must make the following changes to them:

1. Change instances of *DtPvdDefaultFactory* and *DtPvdApplication*, to *DtVrfGuiDefaultFactory* and *DtVrfGuiApplication*.
2. If there is a window class in the example, subclass that window from *DtVrfGuiWindow* rather than *DtPvdWindow*.
3. Add the following libraries to the link line:
 - ♦ *vrftprotocol.lib*
 - ♦ *vrfguieventhandlerprotocol.lib*
 - ♦ *vrfmsgsprotocolMD.lib*
 - ♦ *vrftasksMD.lib*
 - ♦ *vrfutilMD.lib*
 - ♦ *vrfcontrolprotocolMD.lib*
 - ♦ *vrfplanprotocolMD.lib*
 - ♦ *vrfcoreprotocolMD.lib*.

where *protocol* is 5 for DIS or RPR for HLA.

When you rebuild the example, it will look like a VR-Forces example not a Plan View Display example.

4.7. New Architecture for Connecting Components

The mechanism for inter-component communication has been redesigned to make it more flexible, and easier to configure. Under the old approach, components that used *DtPortInterfaces* to connect to other components had to know about the class type of the other component that it was connecting to. This created an undesirable class dependency between the components being connected. This problem no longer exists under the new approach.

The classes used to represent data ports and connections between components have changed. The way in which connections between components are specified in the object parameters database has also changed. Details about the new architecture are in *VR-Forces Developer's Guide*. If you have written code to create components or extend components, or you have modified the object parameters database, you may be affected by this change. This section explains how to migrate to the new architecture for creating component.



The model descriptions in Appendix A, “Component Model Descriptions,” have not been updated to the new port and port group mechanism.

4.7.1. Migrating to the New Port/Port Group Architecture

The *DtPort* and *DtPortInterface* classes are now obsolete. However, they are roughly analogous to some of the new classes.

Table 4-1:

Old Class	New Class
<i>DtPort</i>	<i>DtInputPort</i> (<i>inputPort.h</i>)
	<i>DtOutputPort</i> (<i>outputPort.h</i>)
<i>DtPortInterface</i>	<i>DtInputPortGroup</i> (<i>inputPortGroup.h</i>)
	<i>DtOutputPortGroup</i> (<i>outputPortGroup.h</i>)

In the old approach, components (*DtSimComponent*) that provided data had one or more *DtPorts*, and components that accepted data as input had one or more *DtPortInterfaces*. Under the new architecture, components that provide data may have one or more *DtOutputPorts*, *DtOutputPortGroups* (in a has-a relationship) or both, and components that accept data as input may have one or more *DtInputPorts*, *DtInputPortGroups* (also in a has-a relationship) or both. *DtInputPortGroup* and *DtOutputPortGroup* are collections of ports and provide a convenient interface to manipulate ports as a group.

There are several derived *DtInputPort* and *DtOutputPort* classes, representing the different kinds of data that can be exchanged through a port. For example the *DtIntegerOutputPort* and *DtIntegerInputPort* classes (both declared in *integerIOPort.h*) are used to send and receive integer values, respectively. These specific types of input and output port classes correspond to the old *DtIntegerPort* class. There are analogous *DtInputPort* and *DtOutputPort* derived classes for all of the old derived *DtPort* class types.

The new *DtInputPortGroup* and *DtOutputPortGroup* classes loosely correspond to the old *DtPortInterface* class. They represent collections of either *DtInputPort* or *DtOutputPort* objects. Use of *DtInputPortGroup* and *DtOutputPortGroups* are optional. They simply provide an interface that makes it convenient to test or set the status of a group of ports.

4.7.2. Creating Ports

If you have created or extended components that generate data and send it through *DtPort* data members, replace them with *DtOutputPorts*. For example, if you have created a type of component that has a *DtIntegerPort* data member, replace it with a *DtIntegerOutputPort* (*integerIOPort.h*).

Similarly, if you have created a component that accepts data through its own *DtPortInterface* data member, replace it with the corresponding *DtInputPort* data members, and (optionally) a *DtInputPortGroup*. For example, the *DtAutomotiveActuator* used to have a *DtAutoActuatorPortInterface* data member. It now has the following data members instead:

```
DtAnalogInputPort* myThrottlePort;  
DtAnalogInputPort* mySteeringPort;  
DtAnalogInputPort* myBrakePort;  
DtIntegerInputPort* myGearPort;  
DtBooleanInputPort* myParkingBrakePort;  
DtInputPortGroup* myAutomotiveControlPortGroup;
```

To create your new port data members, override `createPorts()` in your component. In your implementation of `createPorts()`, create your new port, choosing a name for the port that you know will be unique within the component. These names will be used in the parameter database to specify how the component will be connected to another component.

After you create your new port data member, add it to the appropriate list of ports in the component, or to a port group owned by the component. This will enable the port to be looked up and connected to another port by the component manager. Components maintain lists of *DtInputPorts* and *DtOutputPorts*, and *DtInputPortGroups* and *DtOutputPortGroups*. If your new port is not meant to be part of a port group, then just add it to the appropriate list of ports on the component, for example:

```
myTargetListPort = new DtEntityListOutputPort("target-list");  
myTargetListPort->init();  
addOutputPort(myTargetListPort);
```

If your new port is supposed to be part of a port group, then you need to add it to the port group, instead of the component's port list, e.g.

```
mySteeringPort = new DtAnalogInputPort("steering");  
myAutomotiveControlPortGroup->addPort(mySteeringPort);
```

Add your port to the component's port list or to a port group, but not to both.



Port groups are always created before ports, so you can always safely add ports to port groups in `createPorts()`.

To create new port group members, override `createPortGroups()`. In your implementation, create the port group with a name you know will be unique within the component. The name will be used to configure the connection with another component in the object parameters database. After you instantiate the component, add it to the appropriate port group list in the component, for example:

```
myAutomotiveControlPortGroup =  
    new DtInputPortGroup("automotive-control");  
addInputPortGroup(myAutomotiveControlPortGroup);
```

4.7.3. Changes to Components that Send Data through Ports

You must make the following additional changes to components that send data to migrate your code:

- ♦ Replace calls to `DtSimComponent::takeControlOfPortInterface()` with `DtOutputPort::attemptToActivate()`.

Under the old architecture, components took control of a port interface before placing values on a port with a call to the component's `takeControlOfPortInterface()` member function. Replace these calls with a call to the *DtOutputPort's* `attemptToActivate()` member function. This function indicates whether the port connection is active (that is, the recipient will receive data placed on the port – there is no component with a higher priority currently connected). These calls are typically made at the beginning of a component's tick, so it can determine whether or not to compute and set values on its output ports.

- ♦ Replace calls to `DtSimComponent::giveUpControlOfPortInterface()` with `DtOutputPort::deactivate()`.

In the old design, once a component was done providing data in its port, it relinquished the connection to the port interface with a call to `giveUpControlOfPortInterface()`. This call should be replaced with a call to the *DtOutputPorts'* `deactivate()` member function.

- ♦ Replace calls to `DtPortInterface::setOwnerPriority(-1)` with `DtOutputPort::allowDeactivation()`.

Relinquishing ownership of a port connection without disconnecting ('passive disconnect') was done under the old design by calling the `setOwnerPriority(-1)` function in the *DtPortInterface*. This had the effect of moving the currently connected port to the lowest possible priority, while still maintaining the connection. Replace these calls with a call to the *DtOutputPort's* `allowDeactivation()` member function.

Calls to set values in the new *DtOutputPort* classes are identical with those in the old *DtPort* classes.

4.7.4. Changes to Components that Receive Data through Ports and Port Interfaces

You must make the following changes to components that receive data to migrate your code:

- ♦ Replace calls to test for non-NULL `DtPort*` with `DtInputPort::activeConnection()`.

Under the old architecture, a port was connected to a port interface by setting a pointer to it in the port interface. In the component's tick function, the pointer was checked to see if it was valid (that is, non-NULL). If it was a valid pointer, then the component would assume the connection to the port was valid and retrieve values from it. Replace these tests for non-NULL *DtPort* pointers with a call to the corresponding *DtInputPort*'s `active()` member function.

- Replace calls to `DtPortInterface::dataReceived()` with `DtInputPort::dataReceived()`.

Under the old architecture, the `DtPortInterface::dataReceived()` call was used to notify the *DtPort* that the data had been received. Replace this call with a call to the corresponding `DtInputPort::dataReceived()` member function.

Calls to retrieve values in the new *DtInputPort* classes are identical with those in the old *DtPort* classes.

4.7.5. Updating Object Parameters Database Entries

If you have created or modified any entries in the object parameters database, you may need to modify those entries to be compliant with the new Port/Port Group scheme.



We recommend that you back-up the object parameters database file before making changes.

For a given entity entry in the object parameters database, you do the following:

1. In the object parameters database file, remove all attached-to-list entries in the entity's component descriptors. These entries were used to indicate which port interface a given component should be connect to:

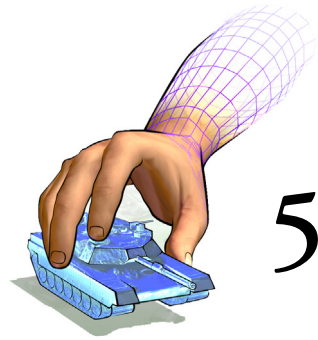
```
(move-to
  (component-descriptor-type "ground-move-to-descriptor")
  (component-type "ground-auto-move-to-controller")
  (attached-to-list
    (powertrain
      (port-interface "automotive-input-port")
    )
  )
  (at-distance 1.0)
  (near-distance 25.0)
)
```

1. To specify how all of an entity's components are connected to each other, add a new connections entry to the entity's parameter entry. It will look something like the following:

```
(connections
  (connect joystick:automotive-control powertrain:automotive-control)
  (connect collision-avoidance:automotive-control
    powertrain:automotive-control)
  (connect move-to:automotive-control powertrain:automotive-control)
  . . .
```

)

Each item in the connections entry specifies a connection between a given port or port group. See section 12.3.2, “Common Elements of Component Descriptors” in the *VR-Forces Users Guide* for details on the format of the connections entry.



Migrating from 3.4 to 3.5

This chapter contains the instructions for moving from VR-Forces 3.4 to VR-Forces 3.5.

5.1. GUI API Migration Instructions

The GUI API has been restructured to accommodate the new tactical overlay features. Parts of the toolkit have also been changed in order to make it easier to add new types of coordinate systems or unit conversions. Examples that rely on this API have been changed to reflect the new API. The sections that follow describe these changes as well as examples you can look at for using the new API.

5.1.1. Removed Classes

The following classes have been removed:

- ♦ *DtObstacleConfigWidget*
- ♦ *DtWaypointConfigWidget*
- ♦ *DtPhaseLineConfigWidget*.

5.1.2. New Classes

The following classes have been added:

- ♦ *DtArchiver* and archivable object classes.
- ♦ *DtPointConfigWidget* encapsulates *DtWaypointConfigWidget* to create any kind of object that would be a single point.
- ♦ *DtTerrainCoordinateSystemCollection* is a collection of coordinate systems. This class is used by all dialog boxes that want to set or display location information. This class is now the central point for adding new types of coordinate system displays. You no longer have to subclass every dialog box that needs to know about a coordinate system. The *windowsCoordinateSystem* example has been updated to show how to add and use a new coordinate system in this context.
- ♦ *DtTMUnitConverterCollection* is a collection of unit converter classes that are used by all dialog boxes and displays that care to translate meters to some other unit. This class is now the central point for adding new types of unit conversions. You no longer have to subclass every dialog box that needs to know about a unit conversion.
- ♦ *DtOverlaySymbolMapper* implements the ability to create overlay objects. *DtMtlSymbolMapper* now subclasses from this object rather than from *DtBaseSymbolMapper*.
- ♦ *DtVrfGuiSymbolMapper* is the new parent for any class that used to subclass from *DtMtlSymbolMapper*.
- ♦ *DtOverlayPublisherHandler* handles the creation and editing of overlay objects.
- ♦ *DtEnvironmentalModelData* handles the data used by overlay objects. *DtControlObjectModelData* classes now subclass from this class. Any symbol updater routine that used to reference *DtControlObjectModelData* now must reference *DtEnvironmentalModelData* instead.
- ♦ *DtTerrainLocationEntryWidget* can be used in dialog boxes that need to enter in positional information. When initialized, this widget can change itself to conform to changes in state of the coordinate system and convert between the current and new system automatically. If you have written code to take this into account, you may be able to replace it with this class. The *imageSplitter* example shipped with the application shows how to use this widget to set positional information to move items on the map area.
- ♦ *DtVrfGuiOverlayPublisherHandler* publishes overlay objects on the network.
- ♦ *DtVrfGuiSymbolUpdater* is the new parent for any class that used to subclass from *DtPvdSymbolUpdater*.

5.1.3. Changes to Existing Classes

The following classes have changed as indicated:

- ♦ *DtAreaConfigWidget* now subclasses from *DtLineConfigWidget*, rather than containing its own UI file.
- ♦ *DtLineConfigWidget* – The initialization function and the create and edit signals have changed.
- ♦ *DtMtlSymbolMapper* no longer takes a *DtMtlEnvironment* as a parameter. You must remove this parameter from any subclass of *DtMtlSymbolMapper* that has the parameter in the constructor and creation function. Also, the creation member function must now return a type of *DtBaseSymbolMapper** rather than a *DtMtlSymbolMapper**.

5.1.4. Migrating Code for Subclasses of *DtMtlSymbolMapper*

You must make the following changes to subclasses of *DtMtlSymbolMapper* in order to conform to the new class structure:

- ♦ The constructor has changed. You no longer need to have the *DtMtlEnvironment** as part of the class, so if your class constructor looked like:

```
MyMtlSymbolMapper(DtMtlEnvironment* env = NULL);
```

it now will look like:

```
MyMtlSymbolMapper();
```

- ♦ The static creator must now return a *DtBaseSymbolMapper** type rather than a *DtMtlSymbolMapper** type:

```
static DtMtlSymbolMapper* create(DtMtlEnvironment* env);
```

changes to:

```
static DtBaseSymbolMapper* create();
```

5.1.5. Migrating Code from *DtLineConfig* and other Config Widgets

The classes that used to create control objects (waypoints, phase lines, and so on) have been changed so that they can create tactical overlay objects. The classes have been organized into line, area and point classes. Any dialog boxes that subclassed from *DtWaypointConfigWidget* should now subclass from *DtPointConfigWidget*. Any class that subclassed from *DtLineConfigWidget* can continue to subclass from *DtLineConfigWidget* (with class updates) and all other classes should subclass from *DtAreaConfigWidget*.

If you used to create your widget by subclassing one of the handlers (line, area, point), you can now simply add the dialog box to be created via the `addDialogCreator()` member function defined in *DtCreateControlObjectHandler*. Whenever an object of the specified type is tried to be created, your dialog box will be instantiated.

You must also update the `init()` member functions and the `createActivated()` and `editActivated()` member function to supply the new parameters. For more information about these member functions and classes, please refer to the GUI API documentation in *VR-Forces Developer's Guide* that describes the new tactical overlay system.

The Minefield example shows how to subclass an existing dialog box and add your new information to it.

5.1.6. Migrating Code that Added Coordinate Systems or Unit Converters to Dialog Boxes

The coordinate system and unit conversion code has been centralized into two classes – *DtTerrainCoordinateSystemCollection* and *DtTMUnitConverterCollection*. These classes hold objects that know how to translate between coordinate systems or units (meters, feet, miles, and so on.)

If you have overridden classes that dealt with coordinate systems or unit converters, you must create a new coordinate system patterned after a *DtTerrainCoordinateSystem* or a *DtTMUnitConverter*. The old member functions and variables have been deprecated.

The `windowsInfoExtension` example shows how to implement a coordinate system so that it can be used in any dialog box that deals with a coordinate system. The widget classes that used to be subclassed to add this capability (the *DtInfoWidget* and *DtBaseEntityInfoWidget*) have been removed.

i

Display of UTM coordinate systems has changed. What used to be the UTM coordinate system for UTM databases (which would show an offset in meters from the origin of the database) now shows the true UTM coordinate of the database as Zone, Easting and Northing values. What used to be displayed under UTM is now displayed under the Database coordinate system, which shows the native coordinates of the database. This means that a Geodetic database shows radians and a UTM database shows meters as an offset from the origin of the database.

5.1.7. Updates to *vrfSim.opd* for Overlay Support

If you are using a customized version of *vrfSim.opd*, you must add the following parameter blocks to support overlay entries:

```
(circle-parameters
  (parameter-type "vrf-base-object-param")
  (object-type 1 (19 -1 -1 -1 -1 -1 -1))
  (category "Circle")
  (bounding-volume
    (local-bvol 0.000000 0.000000 0.000000)
    (offset 0.000000 0.000000 0.000000)
  )
;;      (min-tick-period 1.000) ;; 1 Hz
;;      (min-tick-period-variance 0.500)
```

```

        (local-objects
          (state-repository "vrf-overlay-object-state-repository")
          (net-interface "local-environmental-net-interface")
;;          (net-interface-min-tick-period 0.500)
;;          (net-interface-min-tick-period-variance 0.100)
        )
        (remote-objects
          (state-repository "vrf-overlay-object-state-repository")
          (net-interface "vrf-remote-environmental-net-interface")
;;          (net-interface-min-tick-period 0.500)
;;          (net-interface-min-tick-period-variance 0.100)
        )
      )
      (text-parameters
        (parameter-type "vrf-base-object-param")
        (object-type 1 (20 -1 -1 -1 -1 -1 -1))
        (category "Text")
        (bounding-volume
          (local-bvol 0.000000 0.000000 0.000000)
          (offset 0.000000 0.000000 0.000000)
        )
;;        (min-tick-period 1.000) ;; 1 Hz
;;        (min-tick-period-variance 0.500)
      )
      (local-objects
        (state-repository "vrf-overlay-object-state-repository")
        (net-interface "local-environmental-net-interface")
;;        (net-interface-min-tick-period 0.500)
;;        (net-interface-min-tick-period-variance 0.100)
      )
      (remote-objects
        (state-repository "vrf-overlay-object-state-repository")
        (net-interface "vrf-remote-environmental-net-interface")
;;        (net-interface-min-tick-period 0.500)
;;        (net-interface-min-tick-period-variance 0.100)
      )
    )

    (symbol-point-parameters
      (parameter-type "vrf-base-object-param")
      (object-type 1 (101 -1 -1 -1 -1 -1 -1))
      (category "Symbol")
      (bounding-volume
        (local-bvol 0.000000 0.000000 0.000000)
        (offset 0.000000 0.000000 0.000000)
      )
;;      (min-tick-period 1.000) ;; 1 Hz
;;      (min-tick-period-variance 0.500)
    )
    (local-objects
      (state-repository "vrf-overlay-object-state-repository")
      (net-interface "local-environmental-net-interface")
;;      (net-interface-min-tick-period 0.500)
;;      (net-interface-min-tick-period-variance 0.100)
    )
    (remote-objects
      (state-repository "vrf-overlay-object-state-repository")
      (net-interface "vrf-remote-environmental-net-interface")
;;      (net-interface-min-tick-period 0.500)
;;      (net-interface-min-tick-period-variance 0.100)
    )
  )
)

```

You must also change the state repository type of control objects to be vrf-overlay-object-state-repository. If you do not make this change a warning will be printed to the simulation window and the control object will not be created. The following parameter blocks are examples of an old control object entry and an updated entry:

Old Control Object Entry

```
(generic-area-parameters
  (parameter-type "vrf-base-object-param")
  (object-type 1 (18 -1 -1 -1 -1 -1 -1))
  (category "Area")
  (bounding-geometry-type Area)
  (bounding-volume
    (local-bvol 0.000000 0.000000 0.000000)
    (offset 0.000000 0.000000 0.000000)
  )
;; (min-tick-period 1.000) ;; 1 Hz
;; (min-tick-period-variance 0.500)
(local-objects
  (state-repository "vrf-object-base-state-repository")
  (net-interface "local-environmental-net-interface")
;; (net-interface-min-tick-period 0.500)
;; (net-interface-min-tick-period-variance 0.100)
)
(remote-objects
  (state-repository "vrf-object-base-state-repository")
  (net-interface "vrf-remote-environmental-net-interface")
;; (net-interface-min-tick-period 0.500)
;; (net-interface-min-tick-period-variance 0.100)
)
)
```

New Control Object Entry

```
(generic-area-parameters
  (parameter-type "vrf-base-object-param")
  (object-type 1 (18 -1 -1 -1 -1 -1 -1))
  (category "Area")
  (bounding-geometry-type Area)
  (bounding-volume
    (local-bvol 0.000000 0.000000 0.000000)
    (offset 0.000000 0.000000 0.000000)
  )
;; (min-tick-period 1.000) ;; 1 Hz
;; (min-tick-period-variance 0.500)
(local-objects
  (state-repository "vrf-overlay-object-state-repository")
  (net-interface "local-environmental-net-interface")
;; (net-interface-min-tick-period 0.500)
;; (net-interface-min-tick-period-variance 0.100)
)
(remote-objects
  (state-repository "vrf-overlay-object-state-repository")
  (net-interface "vrf-remote-environmental-net-interface")
;; (net-interface-min-tick-period 0.500)
;; (net-interface-min-tick-period-variance 0.100)
)
)
```

You must make these changes for the following object parameters database entries:

- ♦ generic-point-parameters
- ♦ control-point-parameters
- ♦ target-point-parameters
- ♦ generic-line-parameters
- ♦ phase-line-parameters
- ♦ route-parameters
- ♦ generic-area-parameters
- ♦ control-area-parameters
- ♦ obstacle-parameters.

5.2. Support for Large Terrain Databases

VR-Forces now supports simulation on large terrain databases. Previous versions of VR-Forces were capable of simulating only in UTM (Universal Transverse Mercator) coordinates. Terrain databases were required to be built in UTM coordinates, and the play area could not span more than one UTM zone. (The UTM system divides the earth into 60 vertical slices or UTM zones. Each zone is 6-degrees of longitude in width (about 600km at the equator). Actuators and controllers assumed that constant Z meant constant altitude, and that moving in the positive X and Y directions meant moving east and north respectively.

In order to support large play areas, VR-Forces can now simulate in geocentric coordinates. (In the geocentric coordinate system, the origin is at the center of the earth. The positive X-axis passes through the prime meridian at the equator; the positive Y-axis passes through 90 degrees east longitude at the equator; and the positive Z-axis passes through the North Pole.) Of course, VR-Forces can still simulate in UTM coordinates.

5.2.1. Loading Geocentric Terrain Databases in VR-Forces

VR-Forces can now load terrain databases that are built in geocentric or geodetic (Latitude, Longitude, Altitude) coordinates. When a geodetic database is loaded by the simulation engine, it is automatically converted to geocentric, since it does not make sense to simulate in geodetic coordinates. Conversely, when a geocentric database is loaded by the VR-Forces GUI, it is automatically converted to geodetic coordinates, since it does not make sense for a 2D map to display geocentric coordinates.

In order to avoid the conversion during database loading, you can explicitly load a geodetic or UTM database into the GUI, while instructing the simulation engine (back-end) to load a corresponding geocentric database. See “Creating A Scenario,” in *VR-Forces Users Guide* for the updated procedure.

5.2.2. Geocentric Terrain Databases

There are multiple ways to obtain terrain databases in geocentric coordinates. You can convert OpenFlight terrain databases in geocentric coordinates to MÄK GDB format using *tdbtool*, or import them directly into VR-Forces. You can convert any existing MÄK GDB file from its current coordinate system (UTM, geodetic) to geocentric using *tdbtool* (see “Using the Terrain Converter Tool,” in *VR-Forces Configuration Guide* for details).

Creating Large Area Terrain Databases from DTED Data

Using *tdbtool*, you can create a set of terrain databases from one or more DTED files for simulating over large geographic areas. You can create a geodetic version for viewing in the front-end, and a geocentric version for simulating in the back-end.

To create a large area database from DTED data:

1. Create a *.map* file specifying the DTED file(s) to import. If the DTED data spans multiple UTM zones, set the Projected flag to false (keep the data in geodetic coordinates). Use *tdbTool* to generate a geodetic GDB database.
2. Load the geodetic database into *tdbTool* to generate a geocentric GDB version of the terrain.

5.2.3. Changes to the Terrain API to Support Large Terrain Databases

This section describes the changes to the Terrain API made to support large terrain databases. Additional changes to the Terrain API are described in “[Migrating to the Updated Terrain API](#),” on page 5-13.

Coordinate Systems

VR-Forces maintains the concept of local (database) coordinates. The local coordinate system is the coordinate system used by the terrain database. You can ask for and set an entity position, velocity, acceleration, or orientation in local coordinates. Through version 3.4 of VR-Forces, local coordinates have been synonymous with UTM coordinates, because that was the only terrain database coordinate system supported by VR-Forces. Now local coordinates may be UTM or geocentric coordinates, depending on the coordinate system of the terrain database.

To access the current coordinate system in the back-end:

```
Coordinate_System* coordinateSystem = simManager->terrainDatabase()-  
>coordinateSystem();
```

Converting between Local Coordinates and Other Coordinate Systems

The *Coordinate_System* class (*coordSystem.h*) provides member functions for converting between local coordinates and geocentric, topocentric, and geodetic coordinate systems.

For example, to obtain the altitude of a given position in local coordinates,

```
// Local position may be in geocentric or UTM coordinates, depending on
// the terrain database loaded.
DtVector localPosition;
. . .
DtGeodeticCoord geodeticPosition;
coordinateSystem->localToGeodetic(localPosition, geodeticPosition);

DtInfo("Altitude at local position %s = %f\n",
      localPosition.string().string(), geodeticPosition.alt());
```

Converting local coordinates to topocentric coordinates can be useful for performing 2-D calculations in the topocentric X-Y plane. You can ask the coordinate system for a rotation matrix (*DtDcm*, defined in *LibMatrix.h*) for transforming from the local coordinate system to a topocentric coordinate system at a given location.

For example, to find the downward and projected X-Y components of a local velocity vector:

```
DtVector localPosition;
DtVector localVelocity;
. . .
DtVector topoVelocity;

DtDcm localToTopoDcm;
coordinateSystem->localToTopo(localPosition, localToTopoDcm);

DtVector topoVelocity;
DtDcmVecMul(localToTopoDcm, localVelocity, topoVelocity);

DtInfo("The magnitude of the downward component of velocity is %f\n",
      topoVelocity[2]);
DtInfo("The projected magnitude of velocity (in the x-y plane) is %f\n",
      sqrt(topoVelocity[1]* topoVelocity[1] + topoVelocity[2] *
      topoVelocity[2]));
```

The Gravity Direction Vector

The *Coordinate_System* class has a member function that returns a direction vector pointing in the direction of gravity, at a given local position.

```
DtVector localPosition;
. . .
DtVector downDirectionVec;

coordinateSystem->down(localPosition, downDirectionVec);
```

5.2.4. Other API Changes

Other changes to the API are as follows:

Entity Models

The *DtAutomotiveActuatorComponent* (*autoAct.h*) has been refactored, to make it easier to override specific functionality in *tick()*.

The *DtFixedWingActuator* (*fixWingAct.h*) class has been modified to support its ability to taxi on the ground, take-off, fly, and land.

Interface Messages

VR-Forces no longer specifies position information in database coordinates in interface messages. Positions are always specified in geocentric coordinates, to make it possible to use a different terrain database in the front-end than in the back-end. In particular, it makes it possible to load a geodetic version of a terrain database for display in the front-end, while loading a geocentric version of the terrain in the back-end for simulation.

Remote Control Interface

The remote control interface class, *DtVrfRemoteController* (*vrfController.h*), now requires that geocentric coordinates be passed in to any of the entity or control object creation functions (for example *createEntity()* or *createRoute()*). The underlying interface message class, *DtIfCreateVrfObject* (*ifCrtVrfObj.h*) requires vertices to be specified in geocentric coordinates. The *newScenario()* member function can now take two databases, one for the front-end and one for the back-end.

Removed Classes

DtObstacleManager class has been removed. There are no longer any references to *DtObstacleManager* in the *DtCgf* (*cgf.h*), *DtVrfCreator* (*vrfCreator.h*), and *DtSimManager* (*simMgr.h*) classes.

Deprecated API

In state repository classes (classes derived from *DtVrfBaseObjectStateRepository*), the following functions have been deprecated:

- ♦ *setLocalOrientation()*
- ♦ *localOrientation()*

These functions set and retrieve topocentric Euler angles for orientation. They will work only with UTM databases and have been retained for backward compatibility. They have been replaced by the following coordinate system-independent member functions:

- ♦ *setLocalOrientationWrtDatabase()*
- ♦ *localOrientationWrtDatabase()*

These functions set and retrieve Euler angles with respect to the current local (database) coordinate system. Similar functions for setting and retrieving local orientation in *DtVrfKinematicState* (*vrfKinState.h*) have been superseded by coordinate system-independent versions.

The *DtMissileStateRepository* class (*missileSR.h*) `setStartLocalOrientation()` member function has been deprecated. It sets the initial orientation of the missile in topocentric Euler angles. It will only work with UTM databases. It has been replaced by the member function `setStartLocalOrientationWrtDatabase()`, which specifies the initial orientation of the missile in Euler angles with respect to the local (database) coordinate system.

The *DtVrfObjectGeometry* class (*vrfObjGeom.h*) contains deprecated member functions for doing 2-D calculations, such as determining if a given point is left of a given list of vertices, or if a point lies inside a closed list of vertices. These functions assume all given local positions and local vertices are in UTM coordinates. Coordinate system-independent versions of these functions have been added to *kinTools.h*. They have the same names as those in the *DtVrfObjectGeometry* class, but will work with whatever the current local (database) coordinate system is.

5.2.5. Migrating Your Code to Support Large Databases

With the exception of the *DtAutomotiveActuator* (*autoAct.h*) and *DtFixedWingActuator* (*fixWingAct.h*) classes, if you have extended sensor, controller or actuator classes, they should continue to compile and function correctly on UTM terrain databases. If the only databases you intend to work with are UTM databases, no additional work should be required.

If you want your extended models to work with geocentric databases, they may require additional work to make them work with geocentric terrain databases (depending on the nature of the extensions). You will need to make changes to your models if they contain assumptions about UTM coordinates for an entity's state (position, orientation, velocity, acceleration.)

Getting the Downward Direction Vector

It is no longer appropriate to assume that the negative Z component of a given local position lies along the direction of gravity. This is only true when local coordinates are UTM coordinates. You can get the downward direction vector at a given local position in a coordinate system-independent manner through the *Coordinate_System* class. For example,

```
DtVector localPosition;  
...  
DtVector downDirectionVec;  
  
coordinateSystem->down(localPosition, downDirectionVec);
```

Updating Local Orientation

Replace the following calls to set an entity's local orientation in its state repository (*DtVrfBaseObjectStateRepository*):

- ♦ `localOrientation()`
- ♦ `setLocalOrientation()`

with calls to the following functions (also in its state repository):

- ♦ `localOrientationWrtDatabase()`
- ♦ `setLocalOrientationWrtDatabase()`

The old `localOrientation()` and `setLocalOrientation()` member functions worked with topocentric Euler angles (heading, pitch, and roll). They will only work with UTM coordinates. The new `setLocalOrientationWrtDatabase()` and `localOrientationWrtDatabase()` set and retrieve local Euler angles (Euler angles representing orientation with respect the local (database) coordinate system.)

Terrain Intersections

The *DtTerrainDatabase* class (*terrainDb.h*) provides an interface for terrain intersections with arbitrary chords. In the VR-Forces back-end, we often need to perform terrain intersections pointing downward in the direction of gravity. The `intersectTerrain()` function declared in *kinTools.h* is a convenience function for performing this type of intersection.

5.3. Migrating to the Updated Terrain API

This section describes the changes you need to make to your code to migrate it to the updated Terrain API. The major changes are:

- ♦ You do not need to create vertex or surface lists
- ♦ The specification manager no longer resides outside the vector network
- ♦ Functions for creating common types of nodes are provided and strongly recommended.

All code for doing any of the above should be modified to use the new interfaces.

5.3.1. New Manager Classes

Three new classes, *DtSpecificationManager*, *DtSurfaceManager*, and *DtVertexManager*, replace the *DtSpecificationList*, *DtSurfaceList*, and *DtGdbVertexList* classes. *DtSpecificationList* and *DtSurfaceList* have been deprecated entirely and are no longer supported at all. *DtGdbVertexList* remains supported for other uses, but is no longer used to store the vertices shared by triangles in the terrain database.

The new manager classes are automatically created by their owners, the *DtTerrainDatabase* and *DtVectorNetwork* respectively, during construction. As such, you no longer need to create instances of the list nodes and add them to the terrain database. Additionally, the manager classes automatically manage their own storage needs. You do not have to manage, and possibly reallocate, the memory for these data structures.

Unlike the list classes, the manager classes are not types of *DtGdbNode*. The *DtSurfaceManager* and *DtVertexManager* are members of the *DtTerrainDatabase* and not located in the terrain group node. The *DtSpecificationManager* is a member of the *DtVectorNetwork*, itself now a member of the *DtTerrainDatabase*, and is not a member of the features node.

Additionally, only one instance of the *DtVertexManager*, *DtSurfaceManager*, and *DtSpecificationManager* should be used, as opposed to the multiple list nodes. This is to increase memory efficiency and the ease of use of the API. Triangle and polygons created by the *DtTerrainDatabase* are automatically associated with the member manager classes.

5.3.2. Migrating from List Nodes to the New Manager Classes

The largest difference between the *DtVertex*, *DtSurface*, and *DtSpecificationList* classes and the new manager classes is that the manager classes extend themselves when they require additional storage space. As a result, you no longer need code for allocating a new array, copying over the elements, and resetting the array owned by the list class.

The manager classes are created upon construction of a terrain database. Therefore, there is no need to create an instance and add it to the terrain database. They are already constructed and ready to be used. You can still set them in the terrain database (inside a *DtVectorNetwork* in the case of the specification manager), but this is not recommended as it may cause problems with any IDs that have already been assigned.

Additionally, since only one instance should be used, if you have extended the file readers you must ensure that they are referencing the main terrain database's manager classes and not creating new managers where new list nodes were previously created.

5.3.3. Creating and Adding *DtPolygonIndirects* and *DtTriangleIndirects*

You no longer need to (nor should you) create a new POLYGON_INDIRECT or TRIANGLE_INDIRECT node, add the surface lists pointer and vertex list pointer, and then add the vertices and surfaces.

The interface provided by *DtTerrainDatabase* expects the following input to create polygons and triangles:

- ♦ *DtTriangleIndirect*:
 - 3 vertices
 - 1 surface

i

The vertices and surface will be managed by the terrain database's manager classes and the actual created triangle will only have IDs associated with the managed items.

- ♦ *DtTriangleIndirect* (with IDs):
 - 3 vertex IDs that reference vertices managed by the terrain database's vertex manager.
 - 1 surface ID that references surfaces managed by the terrain database's surface manager.
- ♦ *DtPolygonIndirect*:
 - A *DtGdbVertexList* of the vertices of the new polygon
 - A surface for the new polygon.

i

The vertices and surface will be managed by the terrain database's manager classes and the actual created triangle will only have IDs associated with the managed items.

i

The constructor allocates its own list of vertices and copies the specified vertices. Thus the new node does not take ownership of the specified *DtGdbVertexList*, and, as a result, the caller is responsible for deleting any memory allocated.

- ♦ *DtPolygonIndirect* (with IDs):
 - A pointer to a list of vertex IDs.
 - The number of vertices that the polygon will have.
 - A surface ID

i

The constructor allocates its own list of vertex IDs and copies the specified vertex IDs. Thus the new polygon node does not take ownership of the specified memory, and, as a result, the caller is responsible for cleaning up any memory allocated.

All of the functions mentioned also take two optional parameters:

- ♦ A parent node pointer if the terrain database's root terrain node is not to be the parent of the new triangle node.
- ♦ A Boolean flag which defaults to false specifying whether the node should be added to the beginning or end of the list of children of the parent node.

5.3.4. New Code Examples

For examples of how to restructure code to create triangles and polygons using the API, see the example projects:

- ♦ createTerrainDb.dsw
- ♦ addTerrainFeatures.dsw
- ♦ modifyTerrainDb.dsw

5.3.5. Memory Usage

Several changes were made to the API to improve memory usage and efficiency.

- ♦ *DtTriangleIndirect* no longer store extent information. In order to drastically decrease the size of the *DtTriangleIndirect* data structure, the stored extent information was removed.
- ♦ The new functions for creating polygons and triangles automatically use the terrain database's memory manager.
- ♦ The manager classes implement internal memory management.
- ♦ New functionality for reducing vertex/surface reduction.
- ♦ The function `polygonsToTriangles()` has been renamed `polygonNodesToTriangleNodes()` to more accurately reflect what it has been performing. All three-sided polygons will be converted to triangle nodes. Polygons with greater-than three sides remain polygons. (This may change in an upcoming version, but probably through separate tessellation code.)

Duplicate Vertex/Surface Reduction

The reduction of duplicate vertices and surfaces is not new functionality, but its importance is. When creating triangles and polygons using the direct interface – where actual vertices and surfaces are specified instead of vertex and surface IDs, then the `eliminateDuplicate` functions should be called to ensure that memory is efficiently used. Because of the processing time necessary to remove duplicates, it is recommended that these functions are only called once all the geometry nodes have been created. You should also note that the tolerance taken as a parameter to the `eliminateDuplicateVertices` function should be carefully chosen. All vertices within tolerance distance of each other will be welded into a single vertex.

5.3.6. Optimizing Performance

In order to improve performance, several options are recommended. While not actually new additions to the terrain API, because of several bug fixes and the reorganization of the API, they are worth discussing in briefly.

Spatial Index: The spatial index divides the terrain into spatially organized “buckets” for improved intersection calculation time. The trade-off for the improved efficiency is increased memory usage, but for the default values of 100, 100, 1, the additional memory usage is negligible. Users are strongly encouraged to create a Spatial Index when importing data into GDB format. The default settings are conservative so as to not generate too much memory. However, as a result, for finely tuning performance, users are encouraged to experiment with different settings to find more appropriate values for their specific terrains.

Modification of the Spatial Index

When a spatial index is present, any modifications to the terrain must also be reflected in the spatial index. You have two options:

- ♦ Delete and recreate the spatial index. This is really only viable after very large numbers of modifications, but for small terrain databases, may be acceptable.
- ♦ Modification of “buckets” in the spatial index: Currently, the spatial index only supports manual searching and updating. Users must find the cells that contain the nodes they have changed in particular and update their contents appropriately, as well as any new cells that have been affected. For example, if a user deletes a polygon and adds several new triangles, all references to the polygon in the Spatial Index should be removed and the new triangles should be added, possibly as a single group.

Tree Balancing

Tree balancing attempts to “balance” the tree by reorganizing it into a tree with branching and leafing specified by the user. Ideally, this would result in more efficient organization for intersection querying and modification. As with spatial indexing, users will have to experiment to find a level of balancing appropriate for their terrains. This does not give nearly the improvement in performance as a Spatial Index, but can still result in further improvements.

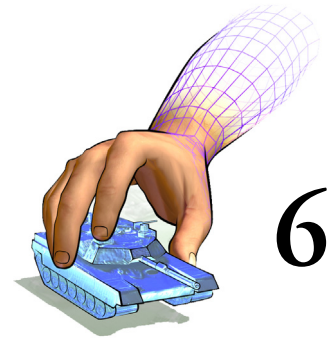
Caveat: A known issue with the Tree Balancing prevents it from being “on” by default. For some terrains, the reorganization of the geometry causes draw errors in the front end, related to polygon ordering (similar to z buffer issues). This does not affect intersection information, and does not affect all terrains. It is primarily an issue with terrains that contain coplanar or nearly coplanar polygons (such as a sea level polygon “underneath” a ground polygon). If any drawing issues occur, discontinue the use of the balancing and focus on improved performance using the spatial index.

Tree balancing may take several hours for a large or complicated terrain.

5.3.7. Deprecated Classes

The following classes are deprecated and should not be used. If not already removed from the Terrain API, they will be soon.

- ♦ *DtTriangleImmediate*
- ♦ *DtPolygonImmediate*
- ♦ *DtGrid*
- ♦ *DtGridPostList*
- ♦ *DtMinefield*
- ♦ *DtRoughTerrain*
- ♦ *DtBodyOfWater*
- ♦ *DtBridge*.



Migrating from 3.5 to 3.6

This chapter contains the instructions for moving from VR-Forces 3.5 to VR-Forces 3.6.

6.1. Changes to the GUI API

The *DtCreateControlObjectHandler* appearance flag has been added as a parameter to the create and edit slots. Member functions that used to look like this:

```
virtual void create(QString name, QString label, QColor color,
    int forceType, bool publish, DtArchivableObject* userData);
virtual void completeEdit(QString name, QString label, QColor color,
    int forceType, bool publish, DtArchivableObject* userData);
```

Now look like this:

```
virtual void create(QString name, QString label, QColor color,
    int forceType, int appearance, bool publish,
    DtArchivableObject* userData);
virtual void completeEdit(QString name, QString label, QColor color,
    int forceType, int appearance, bool publish,
    DtArchivableObject* userData);
```



Bolded code indicates the changes made for this release.

The *DtLineConfigWidget* and *DtPointConfigWidget* classes have been changed to support passing of the appearance flag. Member functions that looked like this:

```
virtual void createActivated(QString name, QString label, QColor color,
    int forceType, bool publish, DtArchivableObject* userData);
virtual void editActivated(QString name, QString label, QColor color,
    int forceType, bool publish, DtArchivableObject* userData);
```

Now look like this:

```
virtual void createActivated(QString name, QString label, QColor color,
    int forceType, int appearance, bool publish,
    DtArchivableObject* userData);

virtual void editActivated(QString name, QString label, QColor color,
    int forceType, int appearance, bool publish,
    DtArchivableObject* userData);
```

The overlay publisher handler has also been changed to support the passing of the appearance flag. Member functions that looked like this:

```
virtual void create(const DtOverlaySymbolDescriptor* desc, QString name,
    QString label, QColor color, QPtrList<Dt3DPoint>& pnts,
    const DtEntityMapperKey& key, int force, bool publish,
    DtArchivableObject* userData);

virtual void edit(QString name, QString label, QColor color,
    QPtrList<Dt3DPoint>& pnts, DtModelData* data, int force, bool publish,
    DtArchivableObject* userData);
```

Now look like this:

```
virtual void create(const DtOverlaySymbolDescriptor* desc, QString name,
    QString label, QColor color, QPtrList<DtPoint>& pnts,
    const DtEntityMapperKey& key, int force, int appearance, bool publish,
    DtArchivableObject* userData);
```

```
virtual void edit(QString name, QString label, QColor color,
    QPtrList<DtPoint>& pnts, DtModelData* data, int force, int appearance,
    bool publish, DtArchivableObject* userData);
```

DtVrfGuiOverlayPublisherHandler has changed to accommodate the addition of the appearance flag. Member functions that used to look like this:

```
virtual void create(const DtOverlaySymbolDescriptor* desc, QString name,
    QString label, QColor color, QPtrList<Dt3DPoint>& pnts,
    const DtEntityMapperKey& key, int force, bool publish,
    DtArchivableObject* userData);

virtual void publishedCreate(const DtOverlaySymbolDescriptor* desc,
    QString name, QString label, QColor color, QPtrList<Dt3DPoint>& pnts,
    const DtEntityMapperKey& key, int force, const QString& parent,
    DtArchivableObject* userData);

virtual void edit(QString name, QString label, QColor color,
    QPtrList<Dt3DPoint>& pnts, DtModelData* data, int force, bool publish,
    DtArchivableObject* userData);
```

Now look like this:

```
virtual void create(const DtOverlaySymbolDescriptor* desc, QString name,
    QString label, QColor color, QPtrList<DtPoint>& pnts,
    const DtEntityMapperKey& key, int force, int appearance, bool publish,
    DtArchivableObject* userData);

virtual void publishedCreate(const DtOverlaySymbolDescriptor* desc,
    QString name, QString label, QColor color, QPtrList<DtPoint>& pnts,
    const DtEntityMapperKey& key, int force, int appearance,
    const QString& parent, DtArchivableObject* userData);

virtual void edit(QString name, QString label, QColor color,
    QPtrList<DtPoint>& pnts, DtModelData* data, int force, int appearance,
    bool publish, DtArchivableObject* userData);
```

6.1.1. Additions to the GUI API

It is now easier to override the creation of entities, aggregates, and control objects. You can override the following member functions to change the behavior of how items are created. The default implementations of these member functions create the items through the remote controller:

- ♦ *DtCreateEntityHandler* now defines the `createEntity()`. Override this member function to do specific entity creation processing.
- ♦ The *DtEntityActionHandler* defines the `createAggregate()` member function. Override this member function to create a specific type of aggregate (this is called as a result of the Aggregate As action).
- ♦ The *DtVrfGuiOverlayPublisherHandler* defines the `sendCreateMessage()` member function. Override this member function to do additional overlay processing.
- ♦ The *DtVrfController* class has been modified to make it easier to capture overlay creation and allow for using custom data to set during create or change.

The `createOverlayObject()` member function used to look like this:

```
virtual void createOverlayObject(  
    const DtEntityType& type,  
    const DtList& vertices,  
    const DtForceType& force,  
    const DtString& name,  
    const DtString& label,  
    int color,  
    const DtString& parentName,  
    const DtSimulationAddress& addr = DtSimSendToAll);
```

The createOverlayObject() member function now looks like this:

```
virtual void createOverlayObject(  
    const DtEntityType& type,  
    const DtList& vertices,  
    const DtForceType& force,  
    const DtString& name,  
    const DtString& label,  
    int color,  
    const DtString& parentName,  
    const DtAppearance& appearance = DtAppearance::nullAppearance(),  
    char* userData = NULL,  
    int dataSize = 0,  
    DtOverlayCreatedCallbackFcn fcn = NULL,  
    const DtSimulationAddress& addr = DtSimSendToAll);
```

The userData that is passed in is copied into internal structures and can be referenced in that structure either by overriding the *DtVrfControllers* processOverlayObjectCreated() member function or by supplying a callback function to be called (which will be called in addition to processOverlayObjectCreated()). The overlay callback function sends through the overlay object name, the entity identifier, and an internal *DtVrfOverlayObjectModify* class, which contains the user information stored in the create call.

Also, the modify message has been changed to include user data. It used to look like this:

```
virtual void sendVrfOverlayObjectModifyMsg(  
    const DtSimulationAddress& addr,  
    const DtEntityIdentifier& id,  
    int color,  
    DtString parentName  
) const;
```

The modify message now looks like this:

```
virtual void sendVrfOverlayObjectModifyMsg(  
    const DtSimulationAddress& addr,  
    const DtEntityIdentifier& id,  
    int color,  
    DtString parentName,  
    char* userData = NULL,  
    int len = 0  
) const;
```

You can override this member function in the *DtVrfController* to process your user data update.

6.2. Changes to how Paper Maps are Drawn

The API for drawing paper maps has changed. To improve performance, all items that used to be treated as QImages are now treated internally as QImages only if the image being loaded from disk has an alpha channel associated with it. If the image has only a black and white mask or no mask at all, it is treated as a QPixmap.

Member functions for returning images to be drawn are no longer called `getScaledPixmap()` or `getScaledImage()`. They are called `getScaledImage()` and either a QPixmap or QImage is passed to each member function to return the image to draw. You can test to see if the particular papermap supports the drawing as an image by calling the `drawAsImage()` member function.

6.3. Terrain API and tdbtool Changes

The Terrain API and tdbtool have the following new features:

- You can clip vector data to the terrain extent and planarize the vector data
- The tdbtool has new options
- *DtTerrainDatabase* has new and changed functions
- VR-Forces ships all libraries or DLLs required to read OpenFlight files
- The *geometryNative* library has been deprecated
- Logging and error checking in the *c7b* loader has been improved
- New class: *DtVectorNetworkIntersectionRecordList*.

6.3.1. Clipping Vector Data to the Terrain Extent

When you import any vector feature data, such as shape, DFD, and DFAD file data, you can clip the vector data to the terrain extent, with options in the map file or using the *tdbtool*.

Clipping the vector data clips all vector features to the terrain to reduce the size of the database as well as remove data that is unwanted.



Area features may be clipped incorrectly if more than one vertex in a row lies outside the clipping region. This will be fixed in an upcoming release. Clipping defaults to true in the map files, so if not specified, it occurs automatically.

6.3.2. Planarizing Vector Data

When you import vector data, you can planarize it with options in the map file or using *tdbtool*. Planarizing flattens the vector data and removes intersections of edges. It may take an extremely large amount of computation time on large vector datasets. Therefore, be careful about planarizing your datasets. Since most vector data other than DFD and DFAD files, such as shape files, is usually well formed, and because of processing and memory concerns, planarizing is not performed by default.

6.3.3. tdbtool Options

tdbtool now has the following options for planarizing vector data and clipping vector data to the terrain extent.

- ♦ `-p tolerance[, excessive_edge_count]` planarizes vector data, where:
 - *tolerance* specifies the required tolerance parameter (0.0001 is usually a good default). The tolerance is used to determine if points are close enough to be considered the same.
 - *excessive_edge_count* stops processing when more than a certain number of edges are generated.
- ♦ `-v` clips vector data to the terrain.

6.3.4. New Map File Options

The following parameters are now supported in ShapeVectorMetafileRecords, DfdD-fadMetafileRecords, and DfadMetafileRecords, or any other vector data metafile records:

- ♦ **Clip {true | false}** – Specifies whether or not to clip vector data to the extents of the polygonal terrain. Default: true.
- ♦ **Planarize {true | false}** – Specifies whether or not to planarize vector data. A planar network is constructed from the vector data and all edge intersections are detected and eliminated. This is an expensive operation and is usually useful only with DFAD and DFD data. Requires the **Tolerance** parameter and optionally takes the **ExcessEdgeCount** parameter. Default: false.
- ♦ **Tolerance tolerance** – Specifies a tolerance to use when deciding if edges intersect one another when planarizing data. If **Planarize** is true, **Tolerance** is required; otherwise, it is meaningless. It is a positive real number. Default: 0.0001.
- ♦ **ExcessEdgeCount count** – Specifies a threshold number of edges, as a positive integer, beyond which planarizing stops. Optional if **Planarize** is true; otherwise, meaningless. Default: 50000.

6.3.5. *DtTerrainDatabase* Functions

DtTerrainDatabase has the following new and changed member functions:

- ♦ `terrainHeightAndSoilType()` – This group of functions provides the height of the terrain at a specified location, either in local (database) coordinates or in latitude and longitude. If there is a successful terrain query at the specified location, the soil type of the polygon at the location is also returned.
- ♦ `closestIntersection()` – This group of functions returns the closest intersection to the specified location. Whereas the `terrainHeight()` functions return the topmost intersection, these functions return the intersection that is closest to the specified location, not necessarily the topmost. It is useful for positioning entities inside areas of the terrain that contain multiple levels, such as inside buildings, or in tunnels under the ground.
- ♦ `intersectVectorNetwork()`, `intersectChordsWithVectorNetwork()`, `intersectLocationsOnVectorNetwork()`, and `allIntersectionsWithVectorNetworkAlongChordList()` – These functions calculate intersections with the vector network according to the specified parameters. They return false if the vector network is empty or if any of the user inputs are NULL.
- ♦ `intersectList()` and `allIntersectsAlongChord()` – These functions now return a bool representing whether any intersection occurred or not. They take parameters that are set to the number of chords with intersections, or the number of intersections along the chord, respectively.

For details see the *DtTerrainDatabase* class description.

6.3.6. The *geometryNative* Library has been Deprecated

To simplify and improve the quality of the Terrain API, the *Dt3DPoint* and *Dt3DExtent* classes have been merged with the *DtPoint* and *DtExtent* classes, respectively. The *DtPoint* class now represents both the deprecated *Dt3DPoint* as well as the original *DtPoint* class. The same is true for the *DtExtent* class. The interface for each class is consistent with the original interfaces provided - the only change is the removal of the `string()` function from the *DtPoint* class. As a result, the *geometryNative* library has been deprecated and contents merged with the appropriate classes in the *geometry* library.

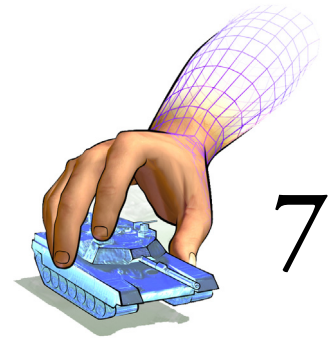
In order to maintain backwards compatibility, the header files *3DPoint.h* and *3DExtent.h* are still distributed with the product. However, they are now empty (do not declare any classes) and they produce a warning when included in a project. They include *point.h* and *extent.h*, respectively, but any references should still be updated appropriately. Instances of *3DPoint.h* or *3DExtent.h* in your files should be changed to be *point.h* or *extent.h* to prevent compilation problems in the future.

Additionally, *Dt3DPoint* and *Dt3DExtent* have been typedefined to be types of *DtPoint* and *DtExtent* inside of *point.h* and *extent.h*. As a result, all references to *Dt3DPoint* and *Dt3DExtent* should compile, but you should update references to these types as soon as possible to prevent problems with future releases.

You should update any references to the *geometryNative* library in any project files or makefiles to reference the *geometry* library instead.

6.4. Miscellaneous API Changes

- Corrected a spelling error in *shapeUtil.h* - `caculatePolygonExtent()` is now `calculatePolygonExtent()`.
- Corrected a spelling error in *taskMsg.h* - `independantly` (used in `myIndependentlyTasked`, `independentlyTasked()`, and `setIndependentlyTasked()`) is corrected to `independently`.
- The *DtFireForEffectOnLocationTask* class (*ffeLocation.h*) now expects locations to be set and retrieved in geocentric coordinates. In previous versions, it expected local (database) coordinates. This was changed to support the capability of loading geocentric databases in the back-end, and a different (geodetic) database in the front-end.



Migrating from 3.6 to 3.7

This chapter contains the instructions for moving from VR-Forces 3.6 to VR-Forces 3.7.

7.1. The Object Parameters Database API and File Format have Changed

The object parameters database file format and API has changed, as follows:

- The object parameters database file format has changed to make editing and maintaining the database easier.
- Object parameters database entries are now loaded on demand, resulting in less memory usage (unused parameter entries are no longer automatically loaded).
- The object parameters database classes have been refactored to remove the restriction that the object parameters database can only be read in from file in *DtReaderWriter* format (the format used by *vrSim.opd*). The interface for loading and saving object parameters database information is now independent of the *DtReaderWriter* format files.

By default VR-Forces continues to use *DtReaderWriter* as the mechanism for serializing parameter data.

7.1.1. New Object Parameters Database Format

The object parameters database has been broken up into multiple files, referenced by a top-level index file. The top-level index file (*vrSim.opd*) contains a list of object parameter entry files (**.ope*), keyed to object type. Each object parameter entry file contains the *DtReaderWriter* representation of a *DtVrfObjectParameters* object. Individual object parameter entry files are self-contained. They contain the complete information about an object.



Parameters specified in *.ope* files do not inherit from more generic object types (as they did in previous releases.) All parameters for an object must be specified in its *.ope* file. Legacy format object parameters databases use the inheritance mechanism as before.

VR-Forces 3.7 includes an object parameters database in the new format (*vrSim.opd*) and in the legacy single-file format (*vrSim_pre3.7.opd*). VR-Forces can read both types of object parameters database file. The legacy file format is included to help you migrate customized object parameters database files to the new format.

Individual parameter entries are loaded on demand. When VR-Forces first creates a particular kind of entity, aggregate, or object, it looks up the filename for the object parameter entry file in the object parameters database and loads it. Any subsequent lookups simply return the entry directly. The object parameters database loads into memory only those parameters necessary for the simulation.

Old format (pre-version 3.7) object parameters database files load all parameters into memory.



Future versions of VR-Forces will not include an object parameters database in the single file format. MÄK strongly encourages customers to migrate their object parameters databases to the new format.

7.1.2. Updating Legacy Object Parameters Database Files

If you have a previous version of VR-Forces and have not edited your object parameters database, then you can run your scenarios with the new default object parameters database, *vrfSim.opd*. If you have edited the default object parameters database, then you should migrate it to the new format using one of the following methods:

- ♦ Convert the object parameters database using the VR-Forces GUI (only in combined mode.)
- ♦ Use the *convertParamDb* utility.
- ♦ Edit the new *vrfSim.opd* file and associated object parameter entry files (*.ope*) by hand.

Converting Object Parameters Databases through the GUI

This option is easiest, but it does not preserve comments when converting from the old format to the new format.



If your object parameters database refers to new kinds of classes, such as new kinds of component descriptors or parameter objects, you must recompile the back-end with your extensions before converting your legacy object parameters database file.

To convert an object parameters database through the VR-Forces GUI:

1. Create a new scenario, or load an existing scenario in combined mode. VR-Forces automatically detects that the object parameters database is in an older format and prompts you to save it in the new format.
2. Click Yes.

Using the *convertParamDb* Utility to Convert Object Parameters Databases

You can use the *convertParamDb* utility to convert-single file object parameters databases into the new format. This tool tries to preserve comments when converting the legacy file into the new format. However, the format of comments may be altered somewhat during the transformation. In particular, comments that start on the same line as a symbol will appear on the line following that symbol in the new format file.



If your object parameters database contains references to any new derived types of component descriptors, resources, or parameter classes, you must rebuild *convertParamDb* before you convert your legacy object parameters database file. Edit *main.cxx* and register creator functions for your new class types before loading the parameter file. For example:

```
DtDefaultResourceFactory myResourceFactory;
DtDefaultComponentDescriptorFactory myComponentDescriptorFactory;
DtDefaultParameterFactory myParameterFactory;

DtSimResource::setFactory(&resourceFactory);
DtComponentDescriptor::setFactory(&componentDescriptorFactory);
DtVrfObjectParameters::setFactory(&parameterFactory);

// Add creator funtions to factories for new customer created
// types in the
// object parameters database here.
//
// e.g.
// parameterFactory.addCreatorFcn(DtSpaceVehicleParamType,
//     DtSpaceEntityParameters::creator);

DtMtlParameterDb parameterDb;
parameterDb.init();

// Load old-style object parameters database file (pre-version
// 3.7)

if(!parameterDb.load(oldFormatParamDbFile))
    . . .
```

To use the *convertParamDb* utility, run it from the *./bin* directory as follows:

```
convertParameterDb old_parameter_DB_file new_parameter_DB_file
```

Edit the Object Parameters Database Files Directly

If your changes to the object parameters database are not too extensive you may want to edit the new object parameters database files by hand. Open the *.opd* or *.ope* files in a text editor and edit them. This option offers you the most control over the format of your comments.



Parameters in individual object parameter entry (*.ope*) files are not inherited from more generic types, so be sure to specify all parameters that you want to change for an object in its *.ope* file.

7.1.3. New Object Parameters Database API

DtVrfParameterDb is the base object parameters database class. *DtVrfParameterDb* holds a collection of *DtVrfObjectParameters* entries, keyed on *DtObjectType*. It has member functions for adding, removing, and looking up *DtVrfObjectParameters* entries. It also includes an interface for *load()* and *save()*, but with an empty implementation. Developers can create derived types of object parameters database classes that know how to load and save themselves from file in whatever format they choose.

VR-Forces has one kind of derived *DtVrfParameterDatabase* type, *DtMtlParameterDb*. *DtMtlParameterDb* extends the base class with the ability to load from and save to *DtReaderWriter* (MTL) format. The *DtMtlParameterDb* class knows how to read and write object parameters database files in this format and in the older, single file format.

The *DtCgf* class is responsible for creating the object parameters database object. By default, it creates an instance of type *DtMtlParameterDb*. Developers can override this behavior and have VR-Forces create some other kind of derived *DtVrfObjectParametersDb*. To do this, register a static creator function for the new kind of *DtVrfObjectParametersDb* with the *DtVrfDtDefaultVrfCreator* class before initializing the *DtCgf*, for example:

```
DtCgf cgf;
. . .
DtDefaultVrfCreator creator;

// Specify the base parameter database type, DtVrfParameterDb. This
// is the simplest kind of parameter database. It does not serialize
// itself to file in MTL format.
creator.setParameterDatabaseCreatorFcn(DtVrfParameterDb::create);

// Initialize the CGF, passing in our altered creator.
cgf->init(exerciseConn, &creator);
```

The new example *createParamDbDynamically* demonstrates how to use a different type of object parameters database with VR-Forces.

7.2. Ground Entities Avoid Obstacles and Features

Ground entities now avoid obstacles drawn on the map. This behavior applies only to obstacles created with the Create → Obstacle menu option. It does not apply to graphical objects drawn on overlays. When an entity detects an obstacle it avoids the obstacle's bounding volume, which is conceptually the smallest rectangle that bounds all of the points in the obstacle.

Ground entities can also avoid features, such as buildings and plants. By default, entities avoid other entities, buildings, water, and vegetation. (LCACs do not avoid vegetation by default.) See the entry for the M9 Ace in the object parameters database for an example of detailed obstruction avoidance configuration.

Notes

- ♦ To specify that ground entities should not move through the terrain (for example, through a wall or through a hill), add the following line to the automotive actuator declaration (powertrain):

```
(check-terrain-collisions True)
```

For performance reasons, this defaults to False, so that entities that do not care about terrain collisions do not perform the check.
- ♦ If any entity is tasked to move to a point that is inside an obstruction, or very close to one, it might never reach the point, but it will continue to try to reach the point as long as the task is in effect. This behavior may also apply to entities following routes that have vertices inside an obstruction.
- ♦ If a terrain has geometric features as well as point vector features, the feature avoidance may not work because the points are placed on the terrain on “top” of the existing geometry, which means they are on top of the geometric feature. A new version of Makland, Makland_OnlyVectorFeatures.gdb, has been provided which does not have point vector features, and as such works with feature avoidance.
- ♦ Path following and obstacle avoidance may not work well together. Entities moving along a road may seek to avoid obstacles, such as trees or light posts, that are placed close to the road, even though they would not hit the obstacles as long as they stay on the road. To prevent this problem from occurring, the ground collision avoidance controller has been given a lower priority than the Move To Path controller.

7.2.1. Configuring Obstruction Avoidance

Obstruction avoidance, formerly obstacle avoidance, has been enhanced in this release. Ground entities can be configured to avoid specific entity types, vector features, or both.



The more vector feature types and entity types specified for avoidance, the more memory and processing power required.

The default parameters for obstruction avoidance have been altered to make them more realistic for avoidance behavior. The **stopping-distance-factor** parameter has been reduced to allow for more aggressive movement and the lateral clearance has been reduced to allow entities to move near each other more easily, especially lifeform entities. If entities are displaying erratic behavior, try raising the values, lowering the speed of the entities, or both.

Configuring Object Types

To add or remove types of objects that an entity type should avoid, add or remove the appropriate object type from the obstruction sensor entry in the object parameter entry file. For example, the following object types are configured for ground entities:

```
(object-type
  (object-type 1 (18 -1 -1 2 -1 -1 -1)) ;; vrfObstacles object type
  (object-type 1 (1 -1 -1 -1 -1 -1 -1)) ;; DtPlatform entity kind
  (object-type 1 (3 -1 -1 -1 -1 -1 -1)) ;; DtLifeForm entity kind
)
```

Configuring Vector Feature Types

Configuration of avoidance of vector features relies on feature specifications in GDB databases. To configure a vector feature type for obstruction avoidance, modify the appropriate **vector-type** parameter in the **vector-obstruction-type** parameter block for the obstruction sensor in an entity's object parameters database entry.

The vector obstruction types are used to create *DtSpecifications*, which are then used by the vector network intersection metrics in determining if obstructions are the requisite types or not. Any attribute specified for avoidance must be in the vector feature for it to be considered an obstruction. Because the vector feature may contain additional properties, simple vector obstruction types can map to a large number of features.

For example, to specify all types of buildings, add a **vector-type** parameter with the single attribute:

```
(vector-obstruction-type
  (vector-type
    (string-attributes
      (DtFeatureGroup "MAK000")
    )
  )
)
```

If you only want to avoid a specific feature, create a more detailed vector-type. For example, the following will cause only standard lights to be considered obstructions.

```
(vector-obstruction-type
  ;; Light Standard (street light) detailed spec
  ;; illustrates how to create a detailed obstruction type
  (vector-type
    (integer-attributes
      (DtFidCode 537)
      (DtFeatureType 0)
      (DtSmc 1)
      (DtSedrisCodes 0)
      (DtOrientation 0)
      (DtHeight 3)
    )
    (string-attributes
      (DtFeatureName "537 Light Standard")
      (DtFeatureGroupName "Power")
      (DtFeatureGroup "MAK050")
      (DtDescription "US-Light Standard/Light Support UK-Light
        Standard/Light Support/Lamp Post")
      (ECC "AL110")
    )
    (real-attributes
      (GDB_Width 0.0)
      (GDB_Length 2.0)
    )
  )
)
```

)

7.3. Saving Default Entity Parameters to the Order of Battle File

In previous releases of VR-Forces (since checkpointing was implemented), when a scenario was saved, most entity parameters were written to the order of battle file, even if their values had not changed from the default values in the object parameters database. If a VR-Forces user subsequently changed the value specified in the object parameters database, it was not picked up by entities that had previously been saved because they got their initial values from the order of battle file. This behavior has changed for VR-Forces 3.7.

In VR-Forces 3.7, when an entity in a scenario is saved to the order of battle file, parameters that are unchanged from the initial values in the object parameters database entry are written out in the form:

```
(parameter-name USE-DEFAULT)
```

The USE-DEFAULT keyword indicates that the next time this file is loaded (as part of a scenario) it should be initialized from data in the entity's object parameters database entry. The practical effect of the USE-DEFAULT keyword in OOB files is that if you modify the object parameters database, then the next time you load a scenario, those changes can affect the entities and objects in the scenario.



You can edit the OOB file manually to override the value obtained from the object parameters database.

This change makes it possible to experiment with changes to entity parameters and view the effects of the changes to existing scenarios, without modifying the order of battle files.

7.4. New Examples

VR-Forces has the following new example applications:

7.4.1. convertParamDb

This example application converts an old format (pre-version 3.7) object parameters database file to the new format. If you have extended VR-Forces by modifying the object parameters database (to add entries for new kinds of entities or to modify existing entries), then you can use this utility to convert your modified file into the new format. See [“Using the convertParamDb Utility to Convert Object Parameters Databases,”](#) on page 7-3 for more information.

7.4.2. createParamDbDynamically

This example demonstrates how to create and populate an object parameters database with entries dynamically, without reading it in from a file. It also shows how to create a type of object parameters database other than the type normally created by VR-Forces.

7.4.3. condition

The condition example (*./examples/condition*) explains how to add a new condition to the VR-Forces plan editor.

7.4.4. showMissileImpact

The Show Missile Impact example (*./examples/showMissileImpact*) shows how to add new symbols and model data to the GUI. This example was written for the MÄK Plan View Display, but it is equally applicable to VR-Forces.

7.5. Collision Detection and Avoidance Has Been Rewritten

The collision avoidance code has been rewritten. The collision detection and avoidance behavior has generally been improved. Entities now have an obstruction sensor, which can be configured to detect different types of entities to avoid. A new collision avoidance controller listens to the sensor and tries to avoid any potential collisions the sensor detects. This is managed programmatically by the Obstruction Manager.

In the object parameters database, you can specify which entities and vector features to avoid.

7.5.1. Component Changes

The collision controller has been split up into a much more extensible set of components.

- ♦ *DtObstructionSensor* (*obstructionSens.h*) – The primary responsibility of the obstruction sensor is to calculate the collision bounding volume of the entity, and to initialize the *DtObstructionManager* and query it for potential collisions. Any potential collisions detected are reported to the collision avoidance controller through ports.
- ♦ *DtObstructionManager* (*obstructionManager.h*) – The Obstruction Manager's responsibility is to determine if the specified collision bounding volume intersects (collides) with any of the user-specified obstructions. Any potential collisions detected are returned to the *DtObstructionSensor*.
- ♦ *DtGroundCollisionAvoidanceController* (*gndColAvoid.h*) – The ground collision avoidance controller is responsible for avoiding potential collisions detected by the obstruction sensor.

7.5.2. API Additions

The following objects have been added to the Terrain API:

- ♦ *DtRwObjectTypeInfo* (*rwObjectTypeInfo.h*) – Used by the *DtObstructionSensor* to allow you to specify an arbitrary number of *DtRwObjectTypes* to treat as obstructions.
- ♦ *DtRwVectorObstructionTypeInfo* (*rwVectorObstructionTypeInfo.h*)
- ♦ *DtRwVectorObstructionType* (*rwVectorObstructionType.h*)
- ♦ *DtRwStringAttribute* (*rwStringAttribute.h*)
- ♦ *DtStringAttribute* (*stringAttribute.h*)
- ♦ *DtRwIntegerAttribute* (*rwIntegerAttribute.h*)
- ♦ *DtIntegerAttribute* (*integerAttribute.h*)
- ♦ *DtRwRealAttribute* (*rwRealAttribute.h*)
- ♦ *DtRealAttribute* (*realAttribute.h*).

These classes are all used by the *DtObstructionSensorDescriptor* to allow users to specify an arbitrary number of *DtRwVectorObstructionTypes*, which are basically human readable *DtSpecifications*. The *DtRwVectorObstructionTypes* are transformed into *DtVectorObstructionIDs* that contain the specification attributes specified in the *DtRwVectorObstructionType*. Any vector features with a matching specification (the attributes of the vector feature may be a superset of the attributes specified in the *DtRwVectorObstructionType*) are treated as obstructions to be avoided by the associated entities.

7.6. Updates to the Move-To Controller Component Hierarchy

Two new controllers have been added to the Move-To controller class hierarchy:

- ♦ *DtGroundMoveToDestinationControllerComponent*
- ♦ *DtGroundMoveToLocationControllerComponent*.

The *DtMoveToDestinationControllerComponent* (*gndMvToDestinationCntlr.h*) is now the base class of the hierarchy, with both the *DtMoveToControllerComponent* and the *DtGroundMoveToLocationControllerComponent* deriving from it. The *MoveToDestination* controller provides all of the functionality necessary for ground entities to move to a specified destination. The destination is stored in the ground entity's process state repository, but is also cached locally. The derived controllers, including not only the *MoveTo* (which is responsible for moving to a control point) and *MoveToLocation*, but also the child classes of the *MoveTo*, are now only responsible for properly updating the location, in world coordinates, stored in the process state repository.

For example, when processing a *MoveToLocation* task, the *MoveToLocation* controller simply calculates the location in world coordinates, sets it in the process state repository, and then defers to the *MoveToDestination* controller for the actual movement to the specified location.

Similarly, when moving to a specified control point (waypoint), the MoveTo controller simply ensures that the location in the process state repository matches the location of the control point. If the location of the control point moves, or if the control point no longer exists, then the controller must update the location in the PSR or clear the task as appropriate.



Note that whenever the location of the destination has changed, the child controllers must invalidate the destination vector so that it is recalculated next time it is requested. This can be accomplished by calling down to the MoveTo controller's `processDestinationChanged()` function, or overriding the function and calling `invalidateDestinationVector()` directly.

7.7. Changes to the *DtGroundMoveToControllerComponent* API

There are several important changes to the MoveTo controller API. Primarily, the following function names have changed:

- ♦ `processWaypointingChanged()` is now `processDestinationChanged()`.
- ♦ `setupControlPoint()` is now `setupDestination()`.

Maintaining backwards compatibility was not possible with these changes.

The MoveToDestination controller has modified somewhat the concept of the goal vector, now the destination vector. The destination vector is cached as before, but now when the destination vector is no longer valid, such as when the location of the destination changes, the controllers call the member function `invalidateDestinationVector()`. This ensures that the next time the destination vector is requested, it is recalculated. In derived controllers, ensure that when performing or responding to any action that invalidates the destination vector, `invalidateDestinationVector()` is called. The easiest way to do this is to override `processDestinationChanged()` to perform any controller specific functionality, and then call up to the parent's version of the `processDestinationChanged()` function.

7.7.1. Migration to the New MoveTo API

If you have derived a controller from any of the MoveTo controller components, you should not have to do much work to update your controllers. Ensure that you are calling the correct functions, `setupDestination()` and `processDestinationChanged()` as appropriate. And ensure that when overriding `processDestinationChanged()`, that you either call up to the parent's version of the function, or call `invalidateDestinationVector()` directly.

Member functions that used the word goal in their names, have had it replaced with destination, for example, `atGoal()`, `passedGoal()`, `nearGoal()`, and so on, have all been changed to `atDestination()`, `passedDestination()`, `nearDestination()`, and so on.

7.8. Changes to the Terrain API

The GDB file format has been updated to accommodate bug fixes and the vector network additions. You are strongly encouraged to re-import (to GDB format) terrain databases from the original source data, especially terrains containing vector data.

The `tdbTool` now lets you specify the notification level for logging. In order to change the level of output logging, use the `-n` option with a value between 0 and 4. See “Using the Terrain Converter Tool,” on page 2-3, in *VR-Forces Configuration Guide* for details.

While the default value for planarizing is true, planarizing is required only for importing DFD and DFAD data. If the DFD and DFAD data is not planarized, results are unpredictable. Shape data does not have to be planarized.

Planarizing a vector network now only detects intersections between linear features (`DtFeatureType` equals 1) and only if the linear features are of MÄK group MAK010 or MAK030, which should be water and roads. As a result, only rivers and road intersections should be detected and fixed.

7.8.1. Deprecated Terrain API Classes

`DtGdbVertexList` class is deprecated and should no longer be used. Use `std::list<DtVertex>` where appropriate.

7.8.2. Updates to the *DtVectorNetwork*

The topology concept has been removed from *DtVectorNetwork*. The elements of the vector network are stored only once, and only in the forward direction. Forward is defined as the direction along the edge from the start point to the next point in the edge. The forward direction represents the direction of traversal for a one-way street, and can be either respected or ignored when traversing the points contained in an edge.

When creating a *DtNetworkEdge* programmatically, there is no longer any need to create an edge in the reverse direction. A single edge should be created.

Additionally, *DtNetworkEdges* now present a point interface, instead of a segment interface. Developers create *DtNetworkEdges* by adding points to them. An edge must have at least two points. Technically, the edges are still represented internally by segments created when points are added. These segments are owned by the *DtNetworkEdge* objects, and not by the *DtVectorNetwork*. However, developers are strongly encouraged to use the point interface, and not access the segments directly as they are now considered an implementation detail and may change in the future.

To iterate over the points of an edge, ask the edge for a *DtNetworkEdgeTraverser* object. When asking an edge to create an edge traverser, you must specify whether or not to respect the directionality of the edge. If you ignore the directionality, both a forward and backward traverser may be created. However, if you respect the directionality, then for a unidirectional street, only a forward edge traverser can be created.

Once a traverser is created for an edge, you must ask for a *DtNetworkEdgePointIter* over that edge. The point iterator acts as expected, and allows you to iterate over the points contained in the edge. The *DtNetworkEdgePointIter* abstracts away the direction of traversal. Thus, when moving both forwards and backwards along an edge, incrementing the *DtNetworkEdgePointIter* returns an iterator pointing to the next point as appropriate.

These changes were made to implement selection of terrain features and to present a much more flexible, robust, and simpler interface to the *DtVectorNetwork* and the *DtNetworkEdge*. See the updated terrain examples for detailed examples of the new vector network API in use, including feature creation and intersection selection.

7.8.3. Clarification of Vector Network Intersection

When the vector network is intersected, the returned features do not necessarily intersect the specified chord if the interest range is larger than the size of the feature as defined by its attributes. A more appropriate conceptualization of vector network intersection is to think of the intersection as selecting the features of the vector network that meet the specified criteria, much like a SQL SELECT statement selects data from a database. An intersection call to the vector network, with a vector record, is conceptually similar to selecting all records from a database where the records' attributes are equal to the vector record.

As a result, to perform an actual intersection with vector features, the features must be selected from the vector network, and then an actual intersection test must be done, using the properties of the features that define an intersectable object, presumably a volume (width, length, height).

7.8.4. Updates to Metrics Used to Calculate Intersections

The *DtRangeNetworkNodeMetric* and *DtRangeNetworkSegmentMetric* classes now respect the `EvalInZ` parameter. If `EvalInZ` is false, they ignore the Z value in their calculations. If `EvalInZ` is true, they evaluate Z when calculating distance.

7.8.5. Updates to the Display of Vector Point Features

Point features that have both length and width are now displayed as solid rectangles. This applies only to DFD data. DFAD point features do not have width attributes.

7.9. GUI API Changes

This section describes changes to the GUI API. Some of these changes require you to update existing code.

7.9.1. Ability to Select Terrain Feature Objects

Symbol selection has been changed to support selection of terrain feature objects. There are now two types of selectors for object selection: *DtObjectSelector* (which takes the place of the existing *DtSymbolSelector*) and *DtTerrainSelector* (which allows the selection of terrain features). If you are currently subclassing *DtSymbolSelector*, you must now subclass the *DtObjectSelector*.

If you have overridden the `selectSymbols()` member function, the prototype has changed to now include the state of the mouse and the state of the keyboard. The symbol selectors are now responsible for checking these flags in order to decide whether or not to merge the new selection into the existing selection. The `merge()` member function is responsible for this operation and will only be called if the Ctrl key is being pressed.

The main member functions for determining whether or not a symbol is valid for a particular selection mode have not changed. If you have added new selection modes or are filtering out objects based on an existing selection mode, once you change your subclass to *DtObjectSelector* you should not have to change any additional code.

7.9.2. The Entity Information Dialog Boxes have been Redesigned

The entity information dialog boxes have been completely rewritten to support the display of resources and to support a smaller, more concise display of information. The dialog boxes are organized as a set of tabbed pages. The pages that are supported for each dialog box types are:

- ♦ *DtEntityInfoDialog*
 - *DtEntityStateInfoPage* – displays information about speed, location, velocity, and so on.
 - *DtBaseEntityInfoPage* – displays entity ID, force ID, entity type and guise.
 - *DtEntityAttributePage* – displays information about entity attributes.
 - *DtResourceInfoPage* – displays resources for the entity and allows for modifications of resource update and display.
- ♦ *DtAggregateInfoDialog*
 - *DtEntityStateInfoPage* – displays information about speed, location, velocity, and so on.
 - *DtAggregateInfoPage* – displays information about number of subordinates, dimensions, and so on.
 - *DtAggregateSubordinateInfoPage* – displays information about subordinates of this aggregate.
 - *DtResourceInfoPage* – displays resources for the entity and allows for modifications of resource update and display.

Migrating Code to the New Entity Information API

If you have modified the aggregate or entity information dialog box, you must update your code to match the new API as follows:

1. Decide which page you want to put your information on.
2. Subclass the specific object for that page and create it in the dialog box you are overriding. Use the correct member function for the dialog box type ([Table 7-1](#)) and return a new instance of your page.

Table 7-1: Entity Information dialog box objects and member functions

Dialog Box Class	Object	Member Function
DtEntityInfoDialog	DtEntityStateInfoPage	createEntityStatePage()
	DtBaseEntityInfoPage	createEntityInfoPage()
	DtEntityAttributePage	createEntityAttributePage()
	DtResourceInfoPage	createResourceInfoPage()
DtAggregateInfoDialog	DtEntityStateInfoPage	createEntityStatePage()
	DtAggregateInfoPage	createAggregateInfoPage()
	DtAggregateSubordinateInfoPage	createAggregateSubordinateInfoPage()
	DtResourceInfoPage	createResourceInfoPage()

3. In your subclass of the page, override `initPageContents()` to add your items to the display. To add an item, call the `addInfoItem()` member function, providing a unique ID and the title of the item. The item is added to the list box that is displayed on the page. The first column displays the title; the second displays the value of the item.
4. To update the value of the item you added, override the `updatePageContents()` member function. The model key and the state repository of the item being updated are passed into this member function. To update a particular item, call `infoItem()` to retrieve the `QListViewItem` created when you added your item. Then pass the unique ID you used when creating the item. Once you have the item, set its value by calling the Qt member function `setText(1, value)` on it.

Adding A New Page to An Information Dialog Box

If you want to create a new page to display items, you must subclass *DtEntityInfoPage* and implement the following member functions:

- `init()`
- `initPageContents()` – adds new page items
- `createPageContents()` – creates the actual page to display information
- `title()` – specifies the title of the tab on the page
- `updatePageContents()` – updates the page's contents.

The `init()` member function should look like this:

```
// The BaseInfoPageUi contains a page that has a list box associated with
// it. If you want to create a new page, this is where the contents should
// go)

QWidget* DtAggregateInfoPage::createPageContents()
{
    myInfoPage = new BaseInfoPageUi(this);

    return myInfoPage;
}

bool DtAggregateInfoPage::init()
{
    if (!DtEntityInfoPage::init())
    {
        return false;
    }

    myInfoView = myInfoPage->InfoBox;
    myInfoView->setSorting(-1);

    return initPageContents();
}
```



The `updatePageContents()` member function is called only if your page is the current page being displayed.

7.9.3. Creating Graphical Objects

The `DtCreateControlObjectHandler::create()` member function has been changed to:

```
virtual void create(QPtrList<DtPoint>& points, QString name,
                  QString label, QColor color, int forceType, int appearance,
                  bool publish, DtArchivableObject* userData, bool createDone = true);
```

The `create()` member function now includes the points that are used to create the object and a new optional flag at the end of the `createDone()` member function. If this flag is true (the default), the create dialog box is removed after the object is created. If this flag is false, the dialog box stays open. This flag was added to support the creation of multiple routes from a terrain selection when turning a selected feature into a route. This change also means that the signals that are emitted from the *DtLineConfigWidget* and *DtPointConfigWidget* have been changed to match the new `create()` member function.

7.9.4. Setting Resources

The set resources widget (*DtSetResourcesWidget*) now queries the back-end for the resources of the selected entity, and displays those resources and the current values. If you have overridden the set resources widget to show a specific resource, you no longer need to override it.

7.9.5. Filtering Pick Lists

The task, set, and condition dialog boxes that require a user to pick an entity or object, such as Move To or EntityInArea, now support filtering of the pick lists. For example, you can filter for all entities, or only friendly entities, and so on. The Move To tasks have been generalized to include all point objects and entities.

These task and set widgets are now subclassed from a new class, *DtSelectionWidget*. This widget is a generic class that allows filtering of entity types. The classes affected by this change are:

- ♦ *DtMoveToWaypointWidget* has been replaced by *DtMoveToWidget*
- ♦ *DtPatrolBetweenWaypointsWidget* has been replaced by *DtPatrolObjectsWidget*
- ♦ *DtEntityInAreaWidget*
- ♦ *DtFireEffectEntityWidget*
- ♦ *DtFollowEntityWidget*
- ♦ *DtMoveAlongRouteWidget*
- ♦ *DtMoveToWidget*
- ♦ *DtPatrolAlongRouteWidget*
- ♦ *DtPatrolObjectsWidget*
- ♦ *DtSetTargetWidget*

If you have subclassed any of these widgets to accomplish the following results, you must update your code, as follows:

- ♦ Modified a widget to add additional information for a task or set data request.
There is now a layout for each of the affected widgets called `SelectionWidgetUi-Layout`. If you have added additional information to a specific layout of one of these widgets, you can make those modifications in the new `init()` member function prototype (`init(DtTMMAPArea, int iNumListBoxes)`).
- ♦ If you have added additional entity types that can be selected for a task or set data request, you can now add them via selection filters. Override the `initFilters()` member function. Call the parent member function to set up the base selection filters, then add your own filters with the `addFilter()` member function. The entity type can be any entity type (specific or wildcard) you want to select. If you want a more complex filter, the following filters are available and you can add them to the filter list by calling `addFilter(DtFilter*)`:
 - *DtObjectFilter* - Select an object of a specific type.
 - *DtNotObjectFilter* - Select any object except this object type.
 - *DtKindLessThanFilter* - Select any object whose `kind()` is less than the kind specified in the object filter (and matching all other conditions).
 - *DtMatchAnyFilter* - Matches any of the supplied entities keys added via the `addKey()` member function.
 - *DtOrFilter* - This filter contains one or more filters such that the first filter that evaluates to true will allow that item to be selected.
 - *DtAndFilter* - This filter contains one or more filters such that all filters must evaluate to true for the particular type in order for this item to be selected.

Creating Filtered Lists

The *DtFilteredListView* class manages filtered lists. You can use this class in your own dialog boxes to allow users to select from a list of filtered entity items. This class also contains a combo box, which, if more than one filter is present, allows the user to select from any of the filters to show a set (or subset) of selectable items. For example, here is the filter code for the *DtMoveToWidget*:

```
bool DtMoveToWidget::initFilters(unsigned int iListBox)
{
    addFilter(trUtf8("Waypoints"),
        DtEntityMapperKey(-1, -1, DtPointObjectKind, -1, -1,
            DtPointObjectCategoryControlPoint, -1, -1, -1));

    DtOrFilter allFilter(trUtf8("All"));

    DtKindLessThanFilter lessThan(DtEntityMapperKey(-1, -1,
        DtLinearObjectKind, -1, -1, -1, -1, -1, -1));
    DtObjectFilter symbols(trUtf8("Overlay Object"),
        DtEntityMapperKey(-1, -1, 101, -1, -1, -1, -1, -1, -1));

    allFilter.addFilter(&lessThan);
}
```

```

allFilter.addFilter(&symbols);

addFilter(trUtf8("Overlay Object"),
DtEntityMapperKey(-1, -1, DtPointObjectKind, -1, -1, -1, -1, -1,
-1));

addFilter(&symbols);

addFilter(trUtf8("Friendly Objects"),
DtEntityMapperKey(DtForceFriendly, -1, -1, -1, -1, -1, -1, -1, -1));

addFilter(trUtf8("Opposing Objects"),
DtEntityMapperKey(DtForceOpposing, -1, -1, -1, -1, -1, -1, -1, -1));

addFilter(&allFilter);

return true;
}

```

The `iListBox` parameter is the list box number that is being initialized (for example, the *DtPatrolObjectsWidget* has two list boxes, each of which can contain their own filter)

7.9.6. Other Changes to the GUI API

The following changes have been made to other objects:

- ♦ *DtSymbol* - A new member function has been added called `scheduleDisplayComplete()` that takes an argument specifying the number of milliseconds this symbol should be displayed. When this member function is called, a `QTimer` is created to run for the specified amount of time. When the timer is finished, the symbol emits the `displayComplete()` signal. You can then connect to this signal to perform some action on it. The detonation and fire symbols use this member function to determine when they should be removed from the display.
- ♦ *DtPvdSymbolGenerator* - There is no longer an event list to determine when fire and detonate interaction symbols are to be removed from display. The new `scheduleDisplayComplete()` member function is used to set up a timer for each symbol to determine when the symbol should be removed from the display. By default this is two seconds for non-animated symbols, or the length of time it takes to complete the animation for animated symbols. If you want to change the length of time these items are displayed, you can call the `setInteractionTimeout()` member function of the symbol mapper for all non-animated interaction symbols or install your own subclass of the symbol mapper and override the `scheduleDisplayComplete()` member function to schedule your own termination time for the symbol
- ♦ *DtPvdMapArea* - The `mySymbolSelector` member variable no longer exists for the map area. There are three new member variables:
 - `myCurrentSelector`
 - `myObjectSelector`
 - `myTerrainSelector`

The `myCurrentSelector` variable is set to either the object or terrain selectors (depending on which one is current). By default it is set to `myObjectSelector`.

- *DtTerrainBackground* - The draw member function now takes an optional painter argument on which to draw. If it receives a `PaintDevice`, it draws to it. If no drawer is supplied, all drawing is done to the `myPixmap` member variable. The `QPainter` argument was added so the background could be drawn to a printer.
- *DtSymbolLensArea* - All references to Quad trees have been removed to clean up the class, because quad trees were not used and not fully implemented.
- *DtPvdEntitySymbol* - A new member variable, `myEmitterSystems`, has been added to keep track of the emitter system symbols that might have been added to an entity. Emitter symbols have their own model data and symbol representation, so this member was added to tie the emitters to their owning entities.

7.10. VR-Forces API changes

This section describes changes to the VR-Forces (back-end) API.

7.10.1. Iterators

The various iterators VR-Forces uses have been changed to use templates. If you use these iterators, you do not need to make any changes to iterate through your lists. Iteration can now be done in a type safe manner. For example, if you have a *DtList* of a certain type of object, you can use the `DtIteratorBase<class T>` iterator to iterate over those objects in type-safe way as follows:

```
DtIteratorBase<MyObject> iterator(myList);
MyObject* objPtr = NULL;

for (objPtr = iterator.first(); objPtr ; objPtr = iterator.next())
{
}
```

7.10.2. Target Detection Sensor

DtTargetDetectionSensor and its use of the *DtVrfObjectStateRepository::isVisible()* member function have been refactored to make it easier to subclass.

The target detection sensor is one of the most complex and execution intensive models in the API. The API changes are designed to simplify the model, while making it faster to run and use. The `isVisible()` member function in the object state repository has been extended to easily allow certain visibility checks to be performed (or not performed) and easily add new types of visibility checks. The following sections describe how the target detection sensor works, the modes it can run in, visibility checks, and the API changes made.

How the Target Detection Sensor Works

The target detection sensor finds targets and places them in a list that other models can use to determine which of the targets the sensor knows about is usable during that tick. When determining which targets can be considered, the target detection sensor:

1. Checks to see if the entity is within range of the sensor.
2. Checks to see if the target is in the sector of responsibility. If it is, it checks the random detection number in the target detection file to see if the target is discovered.
3. If line-of-sight checking is enabled, checks line-of-sight to see if there is anything blocking the target. The three checks currently supported are Polygon, Entity, and Feature.
4. If the entity cannot be seen, but the entity was at one point detected, a random number is drawn to see if the target is removed from the target list. If it is not, it remains in the list even though it cannot be seen.
5. If there is a user target override (Set Target), add that to the list.

The member functions used to perform these steps are as follows:

```
tick()
discoverTargets()
fillInTargetList()
if closest target mode calculateClosestTargets()
acceptTarget()
entityInRange()
targetInSectorOfResponsibility()
checkLineOfSite()
checkEntityDetectForgetProbability()
else checkEntityLists() (uses entity priority lists)
acceptTarget()
entityInRange()
targetInSectorOfResponsibility()
checkLineOfSite()
checkEntityDetectForgetProbability()
```

Specifying the Number of Targets to Track

The updated target detection sensor allows you to specify how many of the closest targets to track over a period of time. This can enhance performance, because the `isVisible()` check (the computationally expensive part of this model) is made only be made for the specified number of closest entities. The following descriptor values have been added to configure this mode:

```
;; (use-closest-targets True) Toggle closest targets mode on or off.
;; (number-closest-targets 10) Specify the number of targets to check.
;; (update-closest-tick-period 10.0) Specify how long to track these
;; targets.
```

If you enable tracking of only the closest targets, the target detection sensor proceeds as follows:

1. At the start of the detection period, the sensor uses the prioritized entity lists to determine the range order of the targets in question.
2. It goes through the range ordered list until it finds the top N closest visible targets.
3. For each tick during the specified period of time, only the N closest targets are checked. If, before the end of the period, a target is destroyed, it is removed from the list and the next closest entity is added to the list.
4. When the `update-closest-tick-period` is over, the target detection sensor repeats the process.

If `use-closest-targets` is false, the sensor always checks the full prioritized entity list each time it is scheduled to run.

DtEntityTargetInformation Object

The *DtTargetDetectionSensor* now puts a more complex object on the target port list than a simple entity ID. This object is called *DtEntityTargetInformation* and contains the following items:

- ♦ The target name
- ♦ The target entity ID
- ♦ The original time of detection (the first time it showed up in the sensor)
- ♦ The last time of detection (the last time the sensor put this entity into the list)

If your code currently iterates through the target port list, you must cast the return value from *DtEntityIdentifier* to *DtEntityTargetInformation*, or you can change your iterator code to the following:

```
DtEntityTargetList::const_iterator iter(*targetList);

for (const DtEntityTargetInformation* info = iter.first(); info;
     info = iter.next())
{
    if (info->id() == currentTargetId)
    {
        targetFound = DtTrue;
    }
}
```

```
        break;
    }
}
```



If you do not update your target detection code, the back-end application will terminate abnormally when your target detection code is exercised.

If you override the functionality to add your own targets to the target list port, you must now add this new type of object to the port as follows:

```
DtEntityTargetInformation target(entity->objectIdentifier(),
    discoveryTime(simTime));

myTargetListPort->addEntity(target);
```

where `simTime` is the current simulation time.

Checking Line-of-Sight

If the target detection sensor is configured to do line-of-sight checking, it uses the `isVisible()` member function in *DtVrfObjectStateRepository*. This member function has been enhanced to allow you to configure which visibility tests to perform and the order in which they are performed. The *vrfSim.mtl* file has a new parameter defined as follows:

```
(setqb VRF_visibilityTestOrder "Polygon Entity Feature")
```

Currently, three modes (polygon, entity, and feature) are supported for visibility testing. The order in which they are listed in the parameter is the order in which the tests are performed. For performance reasons, if your database has a lot of polygons, but few entities, you may want to order the entity check first. If your simulation has a lot of entities, then you should keep the polygon check first.

Using static methods defined in *DtVrfObjectStateRepository*, you can easily add your own visibility tests to the model. These tests apply to all sensors and can be added as follows:

1. To add a new type of test:

```
DtVrfObjectStateRepository::addVisibilityTest(<Test Identifier>,
    DtString("<Test Name>"), <Test Function>);
```

The function callback prototype is:

```
typedef bool (*DtVisibilityTest)(double rangeSquared,
    const DtEntityIdentifier& eyeKey, const DtVector& localEyePoint,
    const DtVector& localEndPoint, const DtVrfObjectStateRepository* sr);
```

The return value of your function will be true if the object is blocked by some item and false if not.

2. To schedule the test, call:

```
DtVrfObjectStateRepository::performTest(Test_Identifier)
```

or you can modify the `VRF_visibilityTestOrder` parameter to include your new test name in the test order.

When you upgrade to the new target detection sensor, you should not have to change much (if any) code, because the main tick() method has not changed. The addition of new entry points, however, should allow you to organize your code to better use the sensor.

7.11. Miscellaneous API Changes

Table 7-2 lists classes that have been deprecated. In many cases, classes were removed to reduce the depth of certain inheritance hierarchies. Obsolete class names have been typedef'd to corresponding new class names to aid in the transition to new names and minimize initial impact on customer code. Users are encouraged to migrate existing code that references deprecated class types and header files to the new types and header files as soon as possible to ensure compatibility with future versions of VR-Forces.

Table 7-2: Deprecated classes

Old Class	New Class
Object Parameters Database Classes	
<i>DtSimObjectParametersDB</i> (objParamDb.h)	<i>DtVrfObjectParametersDB</i> (objParamDb.h)
<i>DtSimObjectParametersList</i> (objParamList.h)	<i>DtVrfObjectParametersList</i> (objParamList.h)
<i>DtSimObjectParameterFactory</i> (paramFact.h)	<i>DtVrfObjectParameterFactory</i> (paramFact.h)
<i>DtSimObjectParameterFactoryEntry</i> (parFactEnt.h)	<i>DtVrfObjectParameterFactoryEntry</i> (parFactEnt.h)
Parameter Classes	
<i>DtSimObjectParameters</i> (simObjParams.h)	<i>DtVrfObjectParameters</i> (vrfObjParams.h)
<i>DtVrfObjectBaseParameters</i> (vrfObjBasePar.h)	<i>DtVrfObjectParameters</i> (vrfObjParams.h)
Miscellaneous Classes	
<i>DtSimObjectSubComponentTypes</i> (simObjSubCmpt.h)	<i>DtVrfObjectSubComponentTypes</i> (vrfObjSubCmpt.h)
<i>DtKeyedSimObjectIterator</i> (keyedObjIter.h)	<i>DtBaseSimObjectIterator</i> (keyedObjIter.h)
<i>DtKeyedVrfObjectIterator</i> (keyVrfObjIter.h)	<i>DtBaseSimObjectIterator</i> (keyedObjIter.h)
State Repository Classes	
<i>DtSimObjectBaseStateRepository</i> (baseObjSR.h)	<i>DtSimObjectStateRepository</i> (simObjSR.h)

Table 7-2: Deprecated classes

Old Class	New Class
<i>DtVrfObjectBaseStateRepository</i> (<i>vrfObjBaseSR.h</i>)	<i>DtVrfObjectStateRepository</i> (<i>vrfObjSR.h</i>)
Vrf Object Classes	
<i>DtSimObject</i> (<i>simObject.h</i>)	<i>DtVrfObject</i> (<i>vrfObject.h</i>)
<i>DtSimObjectData</i> (<i>simObjectData.h</i>)	<i>DtVrfObject</i> (<i>vrfObject.h</i>)
Object Manager Classes	
<i>DtSimObjectManager</i> (<i>simObjectMgr.h</i>)	<i>DtVrfObjectManager</i> (<i>vrfObjMgr.h</i>)
<i>DtSimObjectListInterface</i> (<i>simObjListIF.h</i>)	<i>DtVrfObjectManager</i> (<i>vrfObjMgr.h</i>)

The following classes have been removed:

- ♦ *DtEntityParametersDBCompatibility1x* (*parDbCompat1x.h*)
- ♦ *DtEntityParametersListCompatibility1x* (*parLstCmpat1x.h*).

Table 7-3 lists deprecated header files.

Table 7-3: Deprecated header files

Old filename	New file name
<i>simParamTypes.h</i>	<i>vrfParamTypes.h</i>
<i>simObjTypes.h</i>	<i>vrfObjTypes.h</i>

- ♦ For performance and memory usage reasons, the copy of the parameter entry that used to be made when creating a state repository is no longer made when the state repository is initialized. The **myParameterEntry** member variable points to the original parameter entry in the object parameters database and should only be accessed by the `const DtVrfObjectParameters* parameterEntry() const` member function.

If you want to get a copy of the parameter entry for modification, do not access the **myParameters** entry directly, because it will be NULL on access until a copy is created. This copy is created by calling the non `const` `parameters()` member function to return the **myParameters** entry. If a copy does not exist, one will be created by calling the `createParameters()` member function (at which point you can set the **myParameters** entry to your new parameter instance – this should be the only time you access the **myParameters** member variable directly). The parameters will then be initialized and `setupParameterInheritance()` will be called.

If you try to access the **myParameters** variable directly, the program might crash at runtime because it might be NULL, so your code should be written (or modified) to only call the member function that will return the information you want:

`const DtVrfObjectParameters* parameterEntry() const` returns a `const` pointer to the original parameter entry

`const DtVrfObjectParameters* parameters() const` also returns a `const` pointer to the original parameter entry. This member function is provided for backwards compatibility only and could be deprecated in future releases.

`DtVrfObjectParameters* parameters()` returns a mutable copy of the original parameter entry. Changes to this entry will not effect the original pointer.

- ♦ Due to a performance enhancement change, the **myComponentDescriptor** member variable is now defined to be `const` and all access to this descriptor (including the `componentDescriptor()` member function) has been changed to be `const` access. If you have used this descriptor to modify values within the descriptor (either in the original descriptor or a derived descriptor class), this change will effect all entities that share this descriptor, because this descriptor now points to the original parameter database entry.

If you currently modify member variables in a descriptor class, it is recommended that you make those member variables part of the component that uses them and modify them there. If you must modify the entries in the component descriptor, instead of using the supplied descriptor pointer, you must clone the descriptor

entry and work with the clone. You must also delete the clone in your component destructor to avoid a memory leak.

- ♦ The placement of ground entities has been updated to allow for multiple terrain level support. When a ground entity is first placed on the map, it is placed on the highest possible terrain intersection. After that, it can respond to Set Altitude and Set Location requests that change its altitude. Therefore, for multiple terrain elevations, use Set Altitude and Set Location to change the elevation of an entity. For instance, use these functions to change the floor of a building where the entity is located.

When ground entities move, they apply the same logic when calculating their next position. Thus, if the next calculated location resides in between two levels of terrain, and if the entity is configured to respect terrain intersections when moving and there is no terrain blocking their movement, the entity places itself at the closest elevation to the desired location.

- ♦ A parameter for calculating terrain intersections during movement has been added to the automotive actuator. The parameter, `calculate-terrain-intersections` defaults to false. When set to true, the actuator checks to see if any terrain exists between the current location and the next calculated location. If so, the entity moves only if the intersected terrain is passable. By default, any terrain with a slope greater than 80 degrees is not passable. To override this, derive a new actuator and override the `terrainPassable()` member function. See the class documentation for more information about the new behavior.

7.12. New Build System

The VR-Forces distribution has a new directory — `./scripts`. This directory contains files that allow you to modify example projects and your projects to easily add your own files without having to understand the format of the project files supplied with VR-Forces or to directly modify the system.

Each example or user directory, has the following project-related files:

- ♦ `gen.pro` (the Qt profile that is used to generate the `.dsp` or `.vcproj` files)
- ♦ `files-cc.txt` (a text file that lists the source files in the project)
- ♦ `files-h.txt` (a text file that lists the header files in the project)
- ♦ Possibly a `files-ui.txt` file (Qt designer files that define dialog boxes). This file is present if there are no dialog boxes in the application.

If you want to add dialog boxes to a project that does not have them, you must do the following before you generate your project files:

1. Create a file named `files-ui.txt` that lists the `.ui` files in your project
2. Add the following line to the `gen.pro` file:

```
FORMS = $$system(type $$FILES_UI)
```

To generate a new project:

1. Open up a DOS command window and change your directory to the *.scripts* directory (for example, *C:\MAK\VRF3.7\SCRIPTS*)
2. Add your new source file names to *files-cc.txt*, your new header file names to *files-h.txt* and your new Qt UI files to *files-ui.txt*.
3. Run the *buildProject* program. The syntax for *buildProject* is:

```
buildProject source_directory project_name
```

For example, if you want to modify the *scenarioStatistics* example, run the program as:

```
buildProject examples scenarioStatistics
```

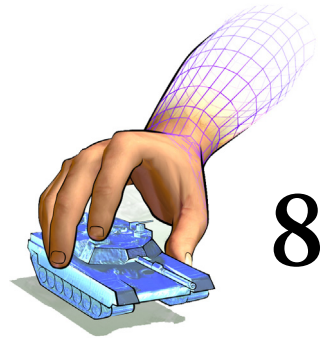
The only exception to this is the *conditionGui* example. To rebuild that project you must run the program as:

```
buildProject examples\condition conditionGui
```

The program creates six files – separate debug and release projects for DIS, HLA13 and HLA1516 projects. The projects named **D.dsp* (or **D.vcproj* if you are using .NET) are debug projects and only the debug configuration of that file is valid. The **R.dsp* (or **R.vcproj* if you are using .NET) are release projects and only the release configuration of that file is valid.

4. Create a new workspace.
5. Add the project files to the workspace that you want to build.

As you modify the *files-cc.txt*, *files-h.txt*, or *files-ui.txt*, remember to regenerate the project files in order to add those files to your project.



Migrating from 3.7 to 3.7.1

This chapter contains the instructions for moving from VR-Forces 3.7 to VR-Forces 3.7.1.

8.1. New Examples

VR-Forces has the following new examples:

- ♦ `envProcessListen` – demonstrates how to listen for and display state updates for environmental process objects (graphical objects) generated by VR-Forces.
- ♦ `addPSR` – demonstrates how to add a process state repository to a component to allow its internal state to be checkpointed to the order of battle file.

8.2. New Configuration Parameters

VR-Forces has new configuration parameters to control how VR-Forces reads messages from the network. The parameters affect the `DtExerciseConn::drainInput()` member function. This function causes VR-Link to read and process input from the network.

- ♦ **VRF_maxDrainInputTime** – sets the maximum time to wait in `drainInput()` before returning. For the DMSO and MÄK RTIs, the default values will drain the entire input stream before returning. For the Pitch RTI, the default values read a single input from the input stream, so set to a non -1 value to process more information. Use with DIS or HLA. Use only with the VR-Forces back-end.
(setqb VRF_maxDrainInputTime -1.0)
- ♦ **VRF_minDrainInputTime** – The minimum time to stay in `drainInput()` (HLA only). Use only with the VR-Forces back-end.
(setqb VRF_minDrainInputTime 0.0)
- ♦ **VRF_maxDrainInputReads** – The maximum number of reads to make in `drainInput()` before returning (DIS only). Use only with the VR-Forces back-end.
(setqb VRF_maxDrainInputReads 100)
- ♦ **PVD_maxDrainInputTime** – sets the maximum time to wait in `drainInput()` before returning. For the DMSO and MÄK RTIs, the default values will drain the entire input stream before returning. For the Pitch RTI, the default values read a single input from the input stream, so set to a non -1 value to process more information. Use with DIS or HLA. Use only with the VR-Forces front-end.
(setqb PVD_maxDrainInputTime -1.0)
- ♦ **PVD_minDrainInputTime** – The minimum time to stay in `drainInput()` (HLA only). Use only with the VR-Forces front-end.
(setqb PVD_minDrainInputTime 0.0)
- ♦ **PVD_maxDrainInputReads** – The maximum number of reads to make in `drainInput()` before returning (DIS only). Use only with the VR-Forces front-end.
(setqb PVD_maxDrainInputReads 100)

8.3. Support for Cruise Missiles

VR-Forces now supports cruise missiles. A cruise missile can fly directly to a target or follow a route.

8.3.1. Changes to the VR-Forces API to Support Cruise Missiles

Cruise missiles are basically a special type of fixed-wing entity. To support them, the fixed-wing components have been modified and new components added.

New Parameters

The Fixed Wing Pilot Controller and all derived controllers, such as move-to, move-along, and follow-entity, have a new parameter called **max-pitch-angle**. It specifies the maximum pitch angle (in radians) which the pilot will use when changing altitudes in this aircraft. This has no relation to actual capabilities of the aircraft model, but only how it will be flown. Default: 0.75.

The Damage Actuator has a new parameter called **terminate-on-destroy**. If set to true, the entity is removed from the simulation as soon as it receives a catastrophic kill. Default: False.

Warhead Detonation Actuator

The warhead detonation actuator, *DtWarheadDetonationActuator*, models the detonation of a warhead that is mounted on an entity, for example, a warhead on a missile. The warhead can detonate based on entity, ground, or water impact, and/or based on proximity to a specific target entity or object. Upon detonation, a detonation interaction is created using parameters passed in using the Set Warhead Info set data request. The host entity is also destroyed and removed from the simulation.

The following parameters are set in the *DtWarheadDetonationActuatorDescriptor*:

- **detonate-on-ground-impact** – If true, a terrain impact with non-water terrain triggers the detonation. Default: True.
- **detonate-on-water-impact** – If true, a terrain impact with water terrain triggers the detonation. Default: True.
- **detonate-on-entity-impact** – If true, any time the host entity's bounding volume intersects with any other entity's bounding volume, the detonation is triggered. Default: True.

The following parameters are configured for an individual detonation actuator by sending the *DtSetWarheadInfo* set data message to the host entity:

- **Warhead Type** – The enumeration of the warhead type to use in the detonation interaction.
- **Fuze Type** – The enumeration of the fuze type to use in the detonation interaction.
- **Attacker Name** – The object name of the attacker to use in the detonation interaction. If specified, the warhead does not detect collisions with the attacking entity. (Presumed to be the launching entity, if this is a munition.)
- **Event Id** – The string representation of the event ID to use in the detonation.
- **Target Name** – The name of the target entity, if proximity detonate is desired.
- **Detonation Proximity** – The proximity to the target that triggers a detonation.

The Warhead Detonation Actuator does not have any ports.

Cruise Missile Launch Controller

The cruise missile launch controller, *DtCruiseMissileLaunchController*, models a launcher of missiles fired only on external orders. The controller responds to *DtFireCruiseMissile* tasks, and creates a new entity of the type specified in its descriptor.

When a *DtFireCruiseMissile* task is received, the controller calls `startTask()` to store the task, and returns. The task is processed by the `tick()` function.

When the controller is ticked:

1. If there is a current task, the simulation is not paused, and the minimum time between launches has elapsed, it continues. Otherwise it returns.
2. It calls `missileResourceAvailable()` to make sure it has not run out of missiles. Returns if this method return false.
3. It calls `consumeMissileResource()` to decrement the missile resource.
4. If calls `createMissileObject()`, which returns a pointer to a newly created simulation object which is the missile being launched.
5. It creates and sends a *DtSetWarheadInfo* message with the host entity specified as the attacker, the target and proximity information from the *DtFireCruiseMissile* task, and the event ID, which is saved for use in the fire interaction created later in the tick function.
6. It calls `createMissilePlan()`, which returns a *DtIfPlan* for the missile. The plan message is then sent to the missile entity. The `createMissilePlan()` function uses the target, route, and speed information in the Fire Cruise Missile task to generate a plan to set the ordered speed, move along the route, then crash into the target.
7. It calls `createAndSendFireInteraction()` to create a fire interaction which is sent to the network.

The following parameters are set in the *DtCruiseMissileLaunchControllerDescriptor*:

- ♦ **munition-type** – The entity type of the missile to be created.
- ♦ **munition-name** – The name of the resource to decrement when a missile is launched.
- ♦ **launch-offset** – The position (relative to the host entity) where the missile will initially be placed.
- ♦ **launch-orientation** – The initial orientation (Tait-Bryan, relative to the host entity) for the missile.
- ♦ **min-time-between-launch** – The minimum simulation time between missile launches.

The Warhead Detonation Actuator does not have any ports.

Fixed-Wing Fly-Into Controller

The Fixed-Wing Fly-Into Controller, *DtFixedWingFlyIntoController*, is a derived type of fixed-wing move-to controller that tries to steer a fixed-wing entity directly into a target entity or object. It responds to *DtCrashInto* tasks. It is designed to hit ground entities from above. This controller contrasts with the typical fixed-wing behavior, in which a fixed-wing entity given a Move To task immediately flies to the altitude of the target point, then moves at that altitude towards the target.

When a *DtCrashIntoTask* is received, the ordered altitude of the fixed wing entity is set to either the current altitude of the entity, or the altitude of the target, whichever is greater.

When this controller is ticked and has an active task it:

1. Checks to see if the target position is within the **direct-fly-range**, or within the **direct-fly-elevation-angle**. Whenever the elevation angle to the target is greater than the **direct-fly-elevation-angle**, it is said to be within the angle.
2. If either of these are the case, the controller calls *flyToDirect()* in the parent *DtFixedWingPilotController* class. This causes the fixed wing entity to orient itself to point directly toward the target location and fly at the ordered speed.
3. If the target object is not in range, the controller calls *flyTo()*, which causes the fixed wing entity to move in the direction of the target while maintaining the ordered altitude.

This controller will never report task complete.

The following parameters are set in the *DtFixedWingCrashIntoControllerDescriptor* (in addition to all *DtFixedWingMoveToController* parameters):

- ♦ **direct-fly-elevation-angle** – The minimum elevation angle from the host entity to the target object which will trigger the direct-flight behavior of this entity.
- ♦ **direct-fly-range** – The minimum range between the host entity and the target object which will trigger the direct-flight behavior of this entity.

The Fixed-Wing Fly-Into Controller has a fixed-wing-control output port.

Changes to Existing Components

The Damage Actuator (*DtDamageActuator*) now checks for firepower-kill and mobility-kill entries in the damage files. These probabilities are specified exactly as the catastrophic-kill entry. The mobility-kill and firepower-kill attributes of the entity are set accordingly.

The Fixed Wing Actuator (*DtFixedWingActuatorComponent*) now calls *fallFromSky()* when the fixed wing entity runs out of fuel or has its mobility-kill appearance attribute set to true.

The Fixed Wing Pilot Controller (*DtFixedWingPilotController*) API has changed slightly with the addition of the `flyToDirect()` member function. You can call `flyToDirect()` instead of `flyTo()`, and it results in different behavior. The `flyTo()` function retains its original behavior of flying toward a target position while maintaining the ordered altitude. The `flyToDirect()` function orients the aircraft to point directly toward the destination location, and fly toward it.

New Tasks and Set Data Requests

VR-Forces has two new tasks and a set data request to support cruise missiles. *DtSetWarheadInfo* (*setWarheadInfo.h*) and *DtCrashIntoTask* (*crashIntoTask.h*) are used programmatically to plan the behavior of a cruise missile after a Fire Cruise Missile task is issued. They are not visible to a user through the GUI. *DtFireCruiseMissileTask* (*fire-CruiseMissileTask.h*) is implemented as the Fire Cruise Missile task in the Task menu of the VR-Forces GUI.

8.4. Copying Parameter Entries

The information in this section was included in VR-Forces 3.7 Release Notes, but did not receive sufficient emphasis. It changes the behavior described in the last sentence of the first paragraph of section 3.3.2 of *VR-Forces Developer's Guide*.

For performance and memory usage reasons, the copy of the parameter entry that used to be made when creating a state repository is no longer made when the state repository is initialized. The `myParameterEntry` member variable points to the original parameter entry in the object parameters database and should only be accessed by the `const DtVrfObjectParameters* parameterEntry() const` member function.

If you want to get a copy of the parameter entry for modification, do not access the `myParameters` entry directly, because it will be NULL on access until a copy is created. This copy is created by calling the non const `parameters()` member function to return the `myParameters` entry. If a copy does not exist, one will be created by calling the `createParameters()` member function (at which point you can set the `myParameters` entry to your new parameter instance – this should be the only time you access the `myParameters` member variable directly). The parameters will then be initialized and `setupParameterInheritance()` will be called.

If you try to access the `myParameters` variable directly, the program might crash at runtime because it might be NULL, so your code should be written (or modified) to only call the member function that will return the information you want:

```
const DtVrfObjectParameters* parameterEntry() const
```

 returns a const pointer to the original parameter entry

```
const DtVrfObjectParameters* parameters() const
```

 also returns a const pointer to the original parameter entry. This member function is provided for backwards compatibility only and could be deprecated in future releases.

```
DtVrfObjectParameters* parameters()
```

 returns a mutable copy of the original parameter entry. Changes to this entry will not effect the original pointer.

8.5. API Changes

In the *DtFixedWingActuatorComponent* class, the data member `myflapsDeployedPort` was changed to `myFlapsDeployedPort`.

An accessor for the *DtExerciseConn* was added to *DtVrfApp*.

The `clamped-frame-rate` parameter has been added to the rotary wing descriptor object (component descriptor type `rotary-wing-component-descriptor`). This parameter is the frame rate threshold in Hertz at which the *DtRotaryWingActuator* will be clamped. When the frame rate drops below this threshold in the actuator, the actuator artificially clamps the frame rate to this value, enabling the model to still function reasonably at low frame rates. The default value for this parameter is 5.0 Hertz. A value of 0.0 Hertz indicates no clamping should be applied.

8.6. Updating Object Parameters Databases

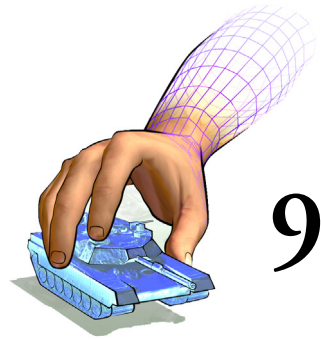
The *VR-Forces 3.7 Release Notes* describe how to update object parameters databases to the new format. This section provides additional details for using the *convertParamDb* utility.

The *convertParameterDb* executable is in the `./bin5`, `./binHLA13`, and `./binHLA1516` directories and not in the `./bin` directory, as documented in the *VR-Forces 3.7 Release Notes*. All three executables work the same way, and you can use any one of them to convert a given parameter database.

The *ammoselect*, *damage*, *detection*, *formation*, and *hit* directories referenced in the old object parameters database file must be available in their original paths (for example, *vr3.6/data/parameters/ammoselect*) during conversion so that they can be copied to the new subdirectory created for the OPE files. Therefore, before running *convertParameterDb*, copy your old-format object parameters database file and the *ammoselect*, *damage*, *detection*, *formation*, and *hit* folders to the `./data/parameters` directory.

The *convertParamDb* utility must be run from the directory that contains the executable. For example, with the current working directory set to *vr3.7/bin5*, a possible command line would be:

```
convertParamDb ../data/parameters/old.opd  
               ../data/parameters/new.opd
```

Migrating from 3.7.1 to 3.8

This chapter contains the instructions for moving from VR-Forces 3.7.1 to VR-Forces 3.8.

9.1. Object Parameters Database Editor

VR-Forces has a new utility, the OPD Editor, a GUI-based editor for the object parameters database. The OPD Editor is in *.bin*. It lets you load your object parameters database and edit parameters for individual objects or for multiple objects within the hierarchy. You can change existing parameter values and add and remove components, resources, and soil types. For a complete explanation of how to use the OPD Editor, see “Using the OPD Editor,” on page 5-22 of *VR-Forces Configuration Guide*.

9.2. Configuration and Startup Changes

This section describes new startup options and changes to configuration files.

9.2.1. Changes to Pathname Interpretation in Object Parameters Database Files

Object parameters database entry files (**.ope*) often contain references to supporting files (such as damage files (*.dmg*) or hit probability files (*.hit*)) that must be loaded. The convention for specifying paths to these supporting files has changed. In previous versions of VR-Forces, the paths to these files were always specified relative to the application path (for example, *.bin5/vrfsim.exe*). Supporting files are now specified relative to the containing file.

The convention was changed to make object parameters database files more self-contained, eliminating the dependency on the application path. You can now move an object parameters database (the top-level *.opd* file and related parameter files) without having to adjust any of the filenames contained in the *.ope* files. Customers have requested this change so that they can maintain custom object parameters databases outside of the VR-Forces file hierarchy and not have to move them every time they install a new version of VR-Forces.

As a result of these changes, the version number for object parameters database files has advanced to version 101. Legacy format files can still be loaded by VR-Forces. However, we recommend that you convert customized object parameters databases to the latest version to ensure compatibility with future versions. Run the *convertParamDb* application to update your legacy format object parameters database (version 100 or pre-3.7.*opd* files) to the latest version.

9.2.2. Configuring Search Paths in Scenario Files

Two new variables have been added to the *vrfsim.mtl* configuration file. The new parameters are:

- **VRF_defaultParameterDatabasePath** – specifies the path to search for *.opd* files when an explicit path is not specified as part of a filename in a scenario file.
- **VRF_defaultTerrainDatabasePath** – specifies the path to search for terrain database files when an explicit path is not specified as part of a filename in a scenario file.

The `VRF_defaultParameterDatabasePath` is used in the search path when loading an object parameters database in the back-end. The object parameters database file is specified in a scenario file. The order in which directories are searched for a given object parameters database are as follows:

1. If the path is absolute, load the file in the given path.
2. If the path is relative, try to load the file relative to the application path.
3. Look in `VRF_defaultParameterDatabasePath`.

The `VRF_defaultTerrainDatabasePath` is used in the search path when loading a terrain database in the back-end. The terrain database file is specified in a scenario file. The order in which directories are searched for a given parameter database are as follows:

1. If the path is absolute, load the file in the given path.
2. If the path is relative, try to load the file relative to the application path.
3. Look in `VRF_defaultTerrainDatabasePath`.
4. If defined, look in the user-defined environmental variable `VRF_TERRAIN_PATH`.

9.2.3. New Startup Options

- ♦ `-F ttl` – specifies the `mcastTtl` value of the exercise connection socket. (DIS only)
- ♦ `-H ethernet_address` – specifies the ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose. (front-end only)
- ♦ The `-R` startup option lets you load a script file at startup.

`-R script_file`

VR-Forces includes the following MTL commands that the GUI can respond to:

- `(opendatabase database_name)` – Opens and displays the specified database as if the user selected File/Open Database and chose a database to open.
- `(loadoverlay overlay_file)` – Loads an overlay file as if the user selected File/Load Overlays.
- `(setlocalextent (list x y z) (list x y z))` – Sets the front end extent (in local coordinates) to the specified North, East, and Southwest coordinates (as if you zoomed into a particular area.)



These commands are only supported locally and are not sent out over the network.

9.2.4. Default Scenario

VR-Forces has a new, user-editable default scenario – *./data/scenarios/default.scn*. This scenario is loaded whenever you start a new scenario. If you want to customize the settings for all new scenarios, edit *default.scn*.

9.2.5. Configuration File Changes

VR-Forces has some new configuration files and has made some changes to existing files, as follows:

- ♦ *vrfGui.mtl* now loads *pvd.mtl* as part of the VR-Forces startup. This is so variables do not have to be duplicated in the two MTL files.
- ♦ The character column in *symbolmap-utf.mtl* file that appears after the font name has been removed.
- ♦ *capabilities.mtl* – stores settings for entity capabilities.
- ♦ *appearance.mtl* – stores settings for entity appearance attributes.
- ♦ *rangeRings.mtl* – configures threat rings for entities.

New Configuration Parameters

VR-Forces has the following new parameters:

- ♦ **VRF_forceParameterDbReload** – in *vrfSim.mtl*, forces *DrCgf* to reload the object parameters database each time a scenario is loaded. This setting defaults to false, which means that if a scenario is started that uses the same object parameters database as the previous one, the object parameters database is not reloaded. When enabled, this setting makes it easier to develop object parameters databases because you can read in changes you have made to the database without having to restart VR-Forces.
- ♦ **PVD_viewControlMessages** – in *pvd.mtl*, enables or disables view control messages. See “[Reflection of the Map View](#),” on page 9-11.
- ♦ **PVD_storeLineValuesMode** – in *pvd.mtl*, specifies whether or not to save the settings for newly created lines to apply to the next new line. See “[Storing Line Settings](#),” on page 9-12.
- ♦ **PVD_defaultFreehandSamplingRate** – in *pvd.mtl*, specifies the default number of points to sample when drawing a freehand mode line. All points are recorded until the object is created, at which point only the sampled amount of points are saved.
- ♦ **PVD_maximumSignalTimeDifference** – in *pvd.mtl*, specifies the number of seconds between when a signal is sent on a certain transmitter frequency and another is sent on the same frequency to try and determine which entities might be responding to an originating signal.
- ♦ **PVD_hideOffScreenSymbols** – in *pvd.mtl*, specifies whether or not to hide symbols when they go off screen instead of destroying them. Setting this parameter to 1 may help with performance for large exercises.
- ♦ **VRF_hostAddrString** – in *vrfGui.mtl*, specifies an ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose.
- ♦ **VRF_defaultParameterDatabasePath** – in *vrfSim.mtl*, specifies the path to search for *.opd* files when an explicit path is not specified as part of a filename in a scenario file.
- ♦ **VRF_defaultTerrainDatabasePath** – in *vrfSim.mtl*, specifies the path to search for terrain database files when an explicit path is not specified as part of a filename in a scenario file.
- ♦ **VRF_logFrameRateStatistics** – in *vrfSim.mtl*, enables (1) or disables (0) logging of frame rate statistics by the back-end. Output is logged to a file named *vrfSimFrameRateStatistics.site number_appnumber_scenario name.timestamp.log*. For example, the following file was generated when running the Berkeley scenario with logging enabled: *vrfSimFrameRateStatistics.1_3001_berkeleydemo.1108945497.log*

9.2.6. Host Address String

You can specify the ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose. To specify the ethernet address, use `VRF_hostAddrString` in *vrfGui.mtl* or the `-H` startup option with *vrfGui*.

9.2.7. Temporary Directories

To prevent conflicts in the temporary directories when running multiple copies of VR-Forces from the same directory, VR-Forces now creates unique temporary directories. The directories are created in the path specified by the `TMP` environment variable. If this variable is not specified on Linux, the `/tmp` directory is used. The `TMP` variable must be specified on Windows. If VR-Forces cannot create a temporary directory, an error message is printed and the application aborts. In the front-end, the temporary directory is created by *DtVrfRemoteController* and starts with the prefix *vrfFe*. In the back-end, the temporary directory is created by *DtCgfDispatcher* and has the prefix *vrfBe*. In both cases, the temporary directory is removed when VR-Forces shuts down.

9.2.8. Changing or Hiding the Background Character

You can now change or hide the character used to create the background color of symbols based on TrueType fonts. When you use TrueType fonts for entity symbols or any other symbols configured for the GUI, the background color, for example, blue or salmon, is determined by a character specified in the configuration file. By default, this is the `@` character. (If you have ever run VR-Forces without installing the TrueType fonts, you may have seen `@` characters displayed instead of the expected entity icons.)

The new configuration option lets you specify a different background character or eliminate it altogether. To specify an alternative background character, add it, in double quotes, to the end of the symbol's entry in the *symbolmap-ttf.mtl* file (or any other configuration file that specifies a "ttf" character. To specify no background, add empty double quotes, as follows:

```
(symbol-map (list 0 -1 -1 3 -1 -1 -1 -1 -1) "ttf" "MapSym NK Sea" "0"
  "") ;; Clears out background character
```

This configuration option is optional. You do not have to update existing symbol map files if you do not need to use it.

9.3. Entity Modeling

This section describes improvements to VR-Forces' modeling.

9.3.1. Path Planning

The new Path Movement component (*DtPathMovementController*) allows ground entities (vehicles and lifeforms, including aggregate entities) to plan routes across terrain when instructed to move to a destination. This component uses a standard A* search algorithm to find the best route for the entity from among a set of evenly spaced routes along a grid, combined with any road vector features that exist in the terrain. The spacing and size of the grid routes, as well as the types of vector features that are considered in the search is customizable through the descriptor for the component in the object parameters database file. Vector feature types that are to be treated as impassable can also be specified in the object parameters database. By default, most ground entities treat all features in the MAK Roads group as roads, and all features in the MAK Water group as impassable.

The evaluation of each segment of the path is done through the *DtPathMetric* class. The type of path metric to use is specified in the descriptor for the component. There is a standard factory for path metrics, and a user can define their own new path metric classes to change the way routes are evaluated for any particular entity type. VR-Forces ships with a standard Slope and Soil Type path metric that is used for all ground vehicles. It considers the soil type along each path section, as well as the average slope of the terrain, and compares these values to the capabilities of the vehicle to determine the cost of transversing the path.

To task an entity to calculate a path before moving, select the Plan Path movement option in the Move To or Move To Location dialog box. Entities must have the Path Movement Controller to respond to this task.

Depending on the complexity of the terrain, the size and spacing of the search grid, the length of the path, and the available processor power of the computer being used, it may take anywhere from under a second to multiple minutes for an entity to calculate its path. The path planning is done in a separate thread from the main simulation, so it will not affect the running of the simulation. While an entity is calculating its path, it does not move.

The best path that is found from the search algorithm is published in the simulation, and displayed on the map. You can prevent routes from being published, in which case they are not displayed. The entity still follows the route.

You can edit the route and the entity will move along the modified route. Once the route is published, the entity moves along it as it would along any other route. When the entity reaches its destination, the route is removed.

In the object parameters database that ship with VR-Forces, all ground vehicles have a path movement controller. This component replaces the *DtGroundMoveToPathController* and *DtGroundPatrolBetweenPathController* components, which have been removed.

Notes

- Because multithreading is used in this component, any VR-Forces scenario that makes use of path planning will not be able to be run in repeatable mode.
- In previous versions of VR-Forces, civilian vehicles were configured to follow paths when moving to a location. In VR-Forces 3.10, when you give a ground vehicle a move to task, you can specify direct movement or path planning. As a result of this change, if you run a scenario created with a previous version of VR-Forces and that scenario has civilian vehicles with plans containing move to tasks, the civilian vehicles will default to direct movement, rather than path following.
- Use of path planning may degrade performance. Performance will be improved if you use spatial subdivisions (see [“Spatial Subdivisions,”](#) on page 9-31).

9.3.2. Road Driving

All entities using the *DtMoveToDestinationControllerComponent* class or its derivatives (all ground vehicles, and dismounted infantry) now have a Stay in Lane mode. You can enable road following in the object parameters database entry for the entity, or through a new Set Stay in Lane set data request.

If Stay in Lane is enabled for an entity, it follows any vector features in the terrain that belong to the MAK road feature group that it encounters as it moves. This means that if the entity's direction of movement is approximately along a road, the entity alters course to drive along the road, offset from the road center by a distance specified in the entity's object parameters database entry. If the road turns, or the direction of movement is no longer along the road, the entity reverts to its usual movement directly toward the destination.

Road driving is different from path planning in that it simply makes vehicles follow the rules of the road when they drive on or near a road. It does not take them out of their way to move along roads. It can be used in conjunction with path planning to keep a vehicle on the right side of the road when it has planned a path using the roads.

The default object parameters database files that ship with VR-Forces disable road driving for all entities except the civilian vehicle to preserve behavior compatibility with previous versions.

9.3.3. Improved Aggregate Behaviors

Aggregate components have been redesigned for version 3.8. The changes to aggregate behavior include:

- ♦ Improved move-along-route behavior. Aggregates no longer cut-corners when following a route.
- ♦ New Move into Formation task. This tasks an aggregate to move into a particular formation, at a given location and heading.
- ♦ Improved set formation behavior. Members of an aggregate no longer try to jockey into position when an aggregate is dragged on the map, issued a Set Location request, or Set Heading request.

Hiding Aggregate Formation Routes

You can hide the temporary routes generated by an aggregate when it is given a route-following task using an option on the View menu.

9.4. Terrain Database Changes

The tdbtool has the following new options:

- ♦ Databases can now be converted to flat earth using the `-t` option.
- ♦ The origin of the database can now be manipulated by using `-setorigin` or `-moveorigin` and specifying the new origin with `-lat <D:M:S>`, `-lon <D:M:S>`, and `-zone <1 - 60>`.
 - `-setorigin` only moves the origin of the database. It does not reproject the database to the new origin (so, basically, moves the local origin.)
 - `-moveorigin` moves the database to the new origin location, changing the coordinates within the database (this is useful for moving a database from one zone to another.)
- ♦ New options have been added for specifying the expected number of vertices, surfaces, specifications expected in an imported terrain (any format). This can result in significant memory and performance savings for very large terrain files. The options are:
 - `-numvertices expected_number_of_vertices`
 - `-numsurfaces expected_number_of_surfaces`
 - `-numspecs expected_number_of_specifications`

Example:

```
tdbtool -numvertices 10000000 -o .\example.gdb  
      .\example.flt
```

- ♦ The Makland terrain database used in the maklanddemo scenario now uses a database that does not have polygonal features (buildings, trees, light poles). The result is that intervisibility will not recognize features unless you turn on feature intervisibility in the Intervisibility Options dialog box. Furthermore, you must increase the range of interest for intervisibility to recognize buildings. When computing line-of-sight for entity fire, the back-end may not adequately recognize features.
- ♦ When you import a terrain database, some default processing is done to remove duplicate vertices and, for non-GDB database, to add a spatial subdivision, if one is not specified.

9.4.1. Flat Earth Support

Flat earth flight files can now be loaded up in the front-end and simulated in the back-end. Previous versions of VR-Forces automatically converted flat earth databases to UTM. They are no longer converted, but are supported as flat earth.

9.4.2. RPF Resolution Keyword Addition

Chapter 2, Working With Terrain Databases, in *VR-Forces Configuration Guide* has added documentation for RPF paper map records. These record types have a new keyword that lets you specify the resolution to use. The default values are “1:1M” for CADRG and “5M” for CIB. The syntax is:

```
Resolution "resolution"
```

9.5. Component Code Generator

The Component Code Generator (*./bin/componentCodeGenerator.exe*) is a utility that assists programmers in developing new *DtSimComponent* classes for VR-Forces entities. It automates much of the work needed to set up a simulation component, and writes out C++ source files which can then be tailored by the developer for their specific needs. For details, see Chapter 5, *Creating New Components*, in *VR-Forces Developer's Guide*.

9.6. Improved Performance

The performance of VR-Forces has been greatly improved when simulating with large numbers of entities, for example a simulation of 1500 ground entities on a relatively complex terrain. To achieve this improvement, spatial organization of both *DtVrfObjects* and the vector feature obstructions has been added. All users should see improved performance of their scenarios, and large simulations will exhibit the most improvement. See “[Spatial Subdivisions](#),” on page 9-31, for more information.

9.7. New GUI Features

This section describes new features in the GUI display.

9.7.1. LOS Filter

The line-of-sight filter allows you to view entity line-of-sight within the boundaries of the filter. You can filter by force type and color the lines by force type.

9.7.2. Reflection of the Map View

You can control the map view of other instances of the MÄK Plan View Display and VR-Forces with the view control feature. You can set any individual instance of these applications to ignore view control messages. You can also enable or disable view control messages with the `PVD_viewControlMessages` MTL variable in *pvd.mtl*.

9.7.3. Freehand drawing

You can now draw freehand lines on overlays. Once you draw a freehand line, it behaves like all other lines (for example, an entity can be tasked to move along it).

When you create a freehand line, you can specify the number of points to create along the line.



Once a line has been down-sampled, the points removed during the down-sampling cannot be recovered.

9.7.4. Storing Line Settings

A new configuration parameter, `PVD_storeLineValuesMode`, lets you specify that new lines get created with the same color, width, sampling, and force settings as the previous line that was created. This variable has the following options:

- ♦ 0 - Do not store any values
- ♦ 1 - Store for freehand lines only
- ♦ 2 - Store for all lines.



This variable only applies to line-based overlay objects and is only valid for the current session.

9.7.5. Specifying Line Width

You can now set the width of individual lines. If no width is specified, the default width is used for the particular object type.

For network representation, the high four bits of the appearance of the object is used to represent this width.

9.7.6. Entity Capabilities

You can set entity capabilities (ammunition supply, fuel supply, recovery, repair) with the Set Capabilities set data request. Default entity capabilities are specified in the *capabilities.mtl* file. Developers can set capabilities using the remote control API.

9.7.7. Setting Appearance Attributes

You can set entity appearance attributes with the Set Appearance set data request. Appearance attributes are stored in the *appearance.mtl* file, and are defined from the DIS standard. The *appearance.mtl* file is responsible for displaying attributes both in the Entity Information dialog box and the Set Appearance dialog box.

The Set Appearance dialog box displays the current values. All changes made to the appearance are immediate and are set in the entity state repository without modification or verification.

See “[Setting Appearance Attributes](#),” on page 9-45 for API changes that support this feature.

9.7.8. Loading the Create Menu Dynamically

You can now load alternative Create menu files (*vrf.ent*) during runtime. This does not affect the overlay portion of the menu, only the portion that creates entities and control objects. You can also specify a create menu file in a scenario file.

9.7.9. Ability to Create Cultural Features

The Create menu includes a new sub menu for cultural features. The menu lets you place neutral force objects that represent objects such as bridges, houses, towers, and so on.

9.7.10. Transmitter and Signal Support

VR-Forces now supports display of reflected transmitters (using the VR-Link *DrReflectedRadioTransmitterList*) and captures signal PDUs/interactions to associate signals with the transmitters that might be sending them. The Display Options dialog box has options to display transmitters, display communication lines, and allow for transmitter selection.

You can view information about transmitters by looking at the Transmitters tab on the Entity Information dialog box. The Transmitter Information dialog box shows detailed information about a specific transmitter and any signals that might have been sent by that transmitter.

The transmitter settings are stored in the *settings.mtl* file, so, if you migrate to a new version of the software, you can copy the *settings.mtl* file to the new version to keep those frequency settings.

A new configuration parameter, *PVD_maximumSignalTimeDifference*, allows you to configure the number of seconds between when a signal is sent on a certain transmitter frequency and another is sent on the same frequency to try and determine which entities might be responding to an originating signal.

9.7.11. Threat Circles (Range Rings)

Threat circles show the potential threat radius of an entity. The threat radius is defined in *rangeRings.mtl* and lets you define multiple threat (range) rings for the weapons an entity might have associated with it. The default is to show a ring with cross hairs to more easily show which entity is associated with which range ring.

Range rings have the following display options:

- ♦ Drawing of transparent circles (this mode will significantly degrade the performance of the GUI.)
- ♦ Displaying of multiple rings leading to the outer ring (in the case of a very large range area, this can help to tie a ring to an entity)
- ♦ Displaying of range (distance) labels for the outer and inner range rings.



On small databases, an entity's range may be beyond the edge of the database display, in which case, you will not see it unless you zoom out.

9.7.12. Object Transparency

All overlay objects can now have transparency associated with them. You apply transparency using a slider on the More tab of the Create/Edit object dialog box. You can set transparency from 0 (opaque) to 100 (fully transparent). Transparency does not affect points and highlighting.

9.7.13. Replacing Entity Icons with Generic Symbols

You can replace entity icons with colored dots. This may increase performance. You can also show/hide entity symbols on a per-entity basis or for a selected group of entities.

9.7.14. Papermap Transparency

A new slider has been added to the Raster Map layers dialog box to make setting of transparency levels easier. Select the layer and move the slider to change the transparency level. If the Apply Changes Immediately check box is checked, any change on this dialog box takes effect immediately.

9.7.15. Stealth Navigation Toolbar

The new Stealth Navigation toolbar lets you control the MÄK Stealth eyepoint and attachment modes, if there is a broadcasting Stealth available. The toolbar sets the mode of the selected Stealth or, if there is only one Stealth broadcasting, that Stealth.

If the Stealth Information dialog box has focus, you can use the keypad to move the Stealth in the same way as by using the keypad in the Stealth.

The Stealth Navigation toolbar requires the use of the MÄK Stealth 5.3 or higher.

9.7.16. Displaying Information About Environmental Objects

The new Environmentals List dialog box lists environmental objects (routes, waypoints, overlays, hazard clouds) that are published on the network. You can show all environmental objects or filter by a particular type.

You can also display information about individual environmental objects.

9.7.17. Fire/Detonate Option Separation

You can now configure the display of fire and detonate lines independently in the Display Options dialog box. The Options menu still has one control for both items. The menu option overrides the individual settings in the Display Options dialog box.

9.7.18. Enabling and Disabling the Display of Fire and Detonate Interactions

The Symbols tab on the Display Options dialog box now lets you enable and disable the display of fire and detonate interaction graphics. If these graphics are disabled, fire and detonation lines are also disabled.

9.7.19. Hazard Cloud Display

The GUI now displays hazard clouds that are generated using a protocol developed by ITT. The clouds can either be shown as ellipses with a radius (for each puff associated with a cloud) or as a bounding volume that encompasses the entire cloud. You can display an image instead of a colored ellipse; however, displaying images will dramatically decrease performance of the front-end.

The Hazard Options dialog box lets you configure the display of hazard clouds. These settings are saved to the *settings.mtl* file.

You can display information about a cloud as a whole and each individual puff that makes up a cloud. Since puffs can be densely packed, only one puff can be selected at any given time.

9.7.20. Shadow Text

A new display option has been added to allow shadowing of text for greater label clarity on symbols.

9.7.21. Ability to Specify A Symbol Map and Create Menu in Scenario Files

You can now specify the symbolmap file and create menu configuration file (alternative to *vrf.ent*) in a scenario file. You can specify them in the Open Database dialog box when you create a scenario, or you can edit the scenario file of an existing scenario. If the parameters are blank, the default files are loaded.

9.8. API Changes

This section describes changes to the VR-Forces APIs.

9.8.1. GUI Remote Control

This section describes the classes that support remote control (for example, loading a database) of Plan View Displays or VR-Forces front-ends by sending MTL commands.

PVD Remote Controller

The *DtPvdRemoteController* class constructs and sends MTL commands over the network to be received by other Plan View Displays or VR-Forces front-ends, for example, Open Database. This class uses the *DtMtlNetCommandDispatcher* class to send and receive MTL style commands that can be parsed and executed by the *DtMtlNetParser*. It wraps the *DtMtlNetParser* class, shielding the caller from having to know the syntax of the underlying MTL commands. It also provides a convenient place to collect all of your MTL commands. *DtPvdRemoteController* implements the Open Database and Reflect commands. The chat example demonstrates how to extend the PVD controller to add a new MTL command.

DtMtlVariable

The *DtMtlVariable* class knows how to create and parse itself for all of the intrinsic MTL types (list, string, fixed and float). It has convenience member functions for adding objects (such as a vector) to a command argument. *DtMtlVariables* are used by *DtMtlCommands* to construct an MTL string that can be parsed locally or sent over the network for parsing by remote Plan View Display and VR-Forces GUIs. Typically, you will create *DtMtlVariable* objects when you are creating a list. To add a variable of a specific type, you use the appropriate *DtMtlCommand* *addVariable()* member functions, described in this section.

The *DtMtlVariable* object is a list of *DtMtlVariable* objects.

To create and add a list MTL variable to a *DtMtlCommand*:

1. Create a new *DtMtlVariable* object.
2. Call *addElement()* on this object to add the list elements.
3. Add the *DtMtlVariable* object to the desired *DtMtlCommand* object by passing it into the *DtMtlCommand* through its *addVariable()* member.



While the *DtMtlVariable* must be allocated by the caller, once it is added to a *DtMtlCommand* object (through the *addVariable()* call), the *DtMtlCommand* object assumes responsibility for deleting the *DtMtlVariable*.

DtMtlCommand

DtMtlCommand is an abstract base class that defines how an MTL command is created. Subclassing and adding member functions for getting and setting variables allows for greater type safety.

You must implement the following member functions in derived classes to fully define a MTL command:

- ♦ `virtual DtString command() const = 0` – A command string used when creating the MTL representation of the command.
- ♦ `virtual void setupVariableList() = 0` – Sets up the variables that will be placed into the command string. It is called from the `commandString()` member function of the base class. You can add variables to the list of variables that will be added to the command by using the `addVariable()` member function.

For each MTL command you want to add to an MTL command dispatcher, add it to the parser as follows:

```
myMtlParser.registerCommand(DtMyMtlCommand().command(), callbackFunc,
                             usrPtr);
```

This registers your new command with your MTL dispatcher so it can be referenced in MTL commands and scripts. Failure to register your command with the dispatcher will result in the command being ignored.

When a callback is invoked, a list of lisp tokens is supplied in the first parameter and the lisp environment in the second:

```
void DtObjPtr callbackFunc(const DtObjPtr& args, const DtObjPtr& env)
```

You can use these arguments in their pure form, or you can restore a command object by passing these arguments to an MTL command constructor. You must create a constructor of this type, calling the base class constructor, if you want to automatically rebuild your command. This can be done as follows:

```
DtMyMtlCommand::DtMyMtlCommand(const DtObjPtr& args, const DtObjPtr&
    env) :
DtMtlCommand(args, env, command())
{
    myInt = (int)variable(0);
    myString = (const char*)variable(1);
}
```

Part of this reconstruction will also be the assignment of the user pointer set when adding this member function. This can be referenced from the `user()` member function of the newly restored command. You can use this member function to call an internal member function from the static callback:

```
void DtObjPtr callbackFunc(const DtObjPtr& args, const DtObjPtr& env)
{
    DtMyMtlCommand command(args, env);

    ((DtMyMtlCommand*)command->user())->process(command);
}
```

DtMtlCommandDispatcher

DtMtlCommandDispatcher is a convenience class that allows you to add registered commands to the MTL environment. It is a wrapper around the *DtMtlEnvironment* class that allows usage of the environment without having to know the lower level MTL calls.

DtMtlNetCommandDispatcher

DtMtlNetCommandDispatcher is a network enabled version of the command dispatcher that you can use to automatically register for network receipt of MTL commands and invoke registered callbacks when those messages are received. To register these commands, register the parser with the MTL command interaction as follows:

```
DtMtlNetCommandDispatcher parser;
parser.registerCommand(DtMyMtlCommand().command(), callbackFunc, usr);
DtMtlCommandInteraction::addParser(&exConn, &parser);
```

When an MTL command is received over the network, the callbackFunc registered will be invoked.

DtPvdMtlCommandDispatcher

DtPvdMtlCommandDispatcher checks to see if the user has decided to ignore messages received over the network.

9.8.2. Loading Script Files at Startup

The `-R` startup option lets you load a script file at startup. The script file can contain commands that have been registered with the default application parser. This parser can be retrieved from the `commandDispatcher()` member function in the *DtPvdAppEventController* object.

9.8.3. Object Transparency

A new class, *DtBaseSymbolDrawer*, has been added to support object transparency. If you have created your own drawer and want to take advantage of transparency drawing, instead of subclassing your drawer from *DtSymbolDrawer*, subclass from *DtBaseSymbolDrawer*. All other code can stay the same.

9.8.4. API Changes to Support Transmitters

The Connection Manager emits the following signals for transmitters:

```
transmitterAdded(const DtReflectedRadioTransmitter&)
transmitterRemoved(const DtReflectedRadioTransmitter&)
transmitterUpdated(const DtReflectedRadioTransmitter&)
```

It emits the following signal when a signal PDU is received:

```
signalInteraction(const DtSignalInteraction&);
```

When a new transmitter is added, an associated *DtTransmitterModelData* is created and a new model data object is added to the internal application model data list. This model data is associated with an owning entity model data. If no owning entity exists, the transmitter is not displayed. If the entity appears later in the simulation, the transmitter will be hooked up to that entity and displayed with that entity.

The symbol generator also emits the following signals when a signal PDU is received from the Connection Manager:

```
// Sent when a signal is first received and associated with a transmitter
signalReceived(const DtTransmitterModelData*,
               const DtSignalInteraction&);

// Sent when a transmitter has stopped transmitting after it had been
// transmitting
stoppedTransmitting(const DtTransmitterModelData*);
```

A new class *DtSignalMap* has been added. A *DtPvdApplication* object owns an instance of this class, and any time a new signal PDU is received it tries to map the signal to a possible sender of a signal on that same frequency. If no signal is matched, the signal is considered the first of that kind on that frequency and any subsequent signals received on that frequency (until a specified time has elapsed) are considered a match for that frequency and a communication link is made in the map. The display then draws a communication line based on this new information.

9.8.5. Symbol Mappers

The `getColorForForce()` and `modifySymbol()` member functions now take a `const DtEntityMapperKey&` as the last parameter.

This parameter will contain the entity type requesting the color.

9.8.6. Classed Removed

- The *DtPvdHelpMenu* has been removed and renamed to be a more generic *DtHelpMenu* included by *tmHelpMenu.h*. The basic functionality and menu items are the same.
- The *DtAboutBox* has been renamed to be *DtPvdAboutBox*. You must now include *pvdAboutBox.h* to override about box functionality.

9.8.7. Changes to the FED File

A new object has been added to the FED file. If you use a customized FED file, you must add the following to it:

```
(class PvdViewControl best_effort receive
 (parameter MtlCommandPdu)
)

(class MtlCommand best_effort receive
 (parameter MtlCommandPdu)
)
```

9.8.8. Object Parameters Database Changes

This section describes changes to the object parameters database.

- There was previously no way to specify the names of both the *DtAutomotivePSR* and the *DtVrfBalGunPSR* used by the *DtIndividualLifeformAcquireComponent* through a component descriptor. This prevented the *DtIndividualLifeformAcquireComponent* from being able to be configured with arbitrary process state repository names. To correct this problem, VR-Forces has a new type of component descriptor, *DtIndividualLifeformAcquireDescriptor*. It includes a new parameter for configuring the name of the *DtVrfBalGunPSR*. You can now configure a *DtIndividualLifeformAcquireComponent* with both a *DtAutomotivePSR* and a *DtVrfBalGunPSR*, as follows:

```
(lifeform-acquire
  (component-descriptor-type "individual-lifeform-acquire-descriptor")
  (component-type "individual-lifeform-acquire-controller")
  . . .
  (process-state-repository-name "automotive-process-state-repository")
  (weapon-process-state-repository-name "turreted-ballistic-gun-
    process-state-
    repository")
  . . .
)
```

- Because of the changes to *DtBoundingVolume* (see [“DtBoundingVolume Migration Guide,”](#) on page 9-23), a zero local bounding volume is no longer appropriate (it results in undefined behavior). As a result, all object parameters database file bounding volume values have been updated to have the default values, (1,1,1), for their *local-bvol* parameter, if that parameter was previously zeroes. Any entry with a zeroed *local-bvol* parameter will use the default values and ignore the zeroed bounding volume, and a warning will be generated. This is a temporary solution, until the bounding volume parameter can be removed from non-volumetric entries.
- Paths in *.ope* files are now relative to the file, not the application. See [“Changes to Pathname Interpretation in Object Parameters Database Files,”](#) on page 9-2 for details.

Migrating Customized Object Parameters Databases

VR-Forces 3.10 has many changes and additions to the object parameters database, including the new path planning, damage, obstruction, and aggregate components. If you are using a customized object parameters database and want to take advantage of these new capabilities, you must merge your changes into the new format. This section describes an approach for migrating customized object parameters databases from previous versions of VR-Forces to the latest format.

Customized object parameters database are likely to be in one of two formats – the single-file format used before VR-Forces 3.7, or multi-file format introduced in VR-Forces 3.7. They are also likely to have two types of customizations – changes to the default parameters values supplied with VR-Forces, or the use of locally developed components.

Converting Pre-VR-Forces 3.7 Object Parameters Databases

To convert pre-VR-Forces 3.7 object parameters databases:

1. Run the utility *convertParamDb* to convert the file to the latest version. If you have added new kinds of descriptor or parameter classes, you must rebuild *convertParamDb* to incorporate your classes, and then run it to convert to the new format. See *VR-Forces 3.7 Release Notes*, *VR-Forces 3.7 Release Notes*, or *VR-Forces Migration Guide* for the procedures for using *convertParamDb*.



The *convertParamDb* utility only converts the format of the object parameters database file. It does not add the new components introduced in VR-Forces 3.10.

2. Manually reconfigure the objects in the object parameters database to use the new VR-Forces 3.10 components.

If your customized file only changes parameter values, it might be easier to identify the changes and edit the default VR-Forces 3.10 object parameters database using the OPD Editor. If you have extended VR-Forces and added new components, you will have to evaluate the scope of your changes to determine if it is easier to convert the object parameters database and update the resulting files with VR-Forces 3.10 changes or start with the VR-Forces 3.10 object parameters database and add your extensions.

Merging Customized Object Parameters Databases that Use the Current Format

If you have a customized object parameters database that uses the current format, to take advantage of the changes introduced in this release, you must merge your object parameters database with the release version of the object parameters database. This will incorporate any changes we have made to the 3.8 version with those extensions you have made in previous releases.

To merge object parameters databases:

1. Identify the *.ope* files that you have customized. Compare them to the release versions using a text editor or a graphical comparison/merge tool such as Araxis Merge, or Beyond Compare. Merge your changes into the new file.
2. For customized *.ope* files that do not have comparable release versions, use the OPD Editor to add, delete, or edit components to incorporate the new VR-Forces features as desired.

9.8.9. Centralized Fire and Detonation Processing

Processing of damage and fire interactions has been centralized to simplify the process and improve performance. VR-Forces has a new actuator and descriptor and a new sensor and descriptor for damage adjudication and under-fire determination and two new classes for centralizing the processing, the *DtDetonationManager* and the *DtFireManager*.

Previously, when determining if an entity was under-fire or taking damage, every entity registered for notification of detonation and fire interactions. Then, when one occurred, each entity analyzed the interaction to see if it was applicable to them, and if so, to process it. This was very inefficient as most entities are not the target of a detonation or close enough to be influenced.

Centralizing the processing removes the need for every entity to process all interactions. It relies on the new manager objects to notify the appropriate entity or entities when an interaction involving them occurs. For some types of fire interactions, all entities must still be notified to determine if they are under fire. But for other types, this results in significant improvements.

The new components work together to minimize processing of interactions. The *DtFireManager* registers for both fire and detonation interactions, and the *DtDetonationManager* registers for detonation interactions. Thus the analysis of the interactions for relevance occurs in a single place. The new components – the *DtDamageAdjudicationActuator* and *DtUnderFireDeterminationSensor* register themselves and their entities, and their appropriate callback functions with the new managers. When a detonation or fire interaction occurs, the fire and detonation managers determine which registered entities are affected, and notify only those entities of the interaction. Thus only the appropriate registered entities need process the interaction. There are also the appropriate derived versions of these classes for individual lifeform entities.

The new under fire determination sensor has an added parameter in its descriptor: **under-fire-from-friendly**, which specifies whether or not an entity will consider itself under fire from friendly entities.

To maintain backwards compatibility, the existing components should be used, as they have been updated with bug fixes and the new **under-fire-from-friendly** capability. If you have a customized object parameters database and want to use the new classes, update your object parameters database files to use the new actuator and sensor: damage-adjudication-actuator and under-fire-determination-sensor respectively (or individual-life-form-damage-adjudication-actuator for lifeform entities). If you have derived classes from the old damage actuator, either continue to use the original, or derive your components from these new classes.

The object parameters database files that ship with VR-Forces have been updated to use the new components.

9.8.10. *DtBoundingVolume* Migration Guide

The bounding volume class has been extensively refactored for several reasons:

- ♦ To fix several bugs in the design and implementation of the class.
- ♦ To provide an intuitive and correct interface to the class.
- ♦ To improve the general performance of both the class and of other areas of the simulation that used the class.

As such, the interface and usage of the class has changed significantly. This section describes the new interface for the *DtBoundingVolume* class as well as how to migrate existing code to the new interface. This section attempts to cover the most general usage of the *DtBoundingVolume* class, and thus does not document all changes. However, the class documentation has been extensively updated to explain the structure, interface, and usage of the class. For more detailed information, please look in the class documentation.

Construction

The constructor has changed to take only the necessary elements to create a *DtBoundingVolume* – the half length, width, and height of the lattice, and potentially the length, width and height of the lattice offset as well, in either vector form or component form. Additionally, a *DtBoundingVolume* can now be constructed or assigned from another *DtBoundingVolume* instance.

The *DtDcm* class is no longer used with regards to the *DtBoundingVolume* class. If you have code that is initializing the bounding volume using a *DtDcm*, pull out the diagonal components of the *DtDcm* and use them individually as the parameters for the constructor. Better yet, refactor the code to remove the need for the *DtDcm* at all, and simply pass around the construction parameters directly, for example:

Previously:

```
DtDcm entityDcm;
const DtBoundingVolume* entityBvol = entity()->vrfState()-
    >boundingVolume();
entityBvol->GetLocalBoundingVolume(entityDcm);

DtBoundingVolume* tempBoundingVolume = 0;
tempBoundingVolume = new DtBoundingVolume(entityDcm, entityBvol-
    >GetBvOriginOffset());
```

Now:

```
const DtBoundingVolume& entityBvol = entity()->vrfState()-
    >vrfObjectGeometry()->localBoundingGeometry().boundingVolume();

DtBoundingVolume* tempBoundingVolume = 0;
tempBoundingVolume = new DtBoundingVolume(entityBvol);
```

Alternatively, when setting the entire bounding volume, for initialization, and so on, the operation can now be done as follows:

Previously:

```
// Set the bounding volume
boundingGeometry.setBoundingVolume(boundingBoxVolumeDiagonal);
```

Now:

```
// Set half dimensions.
boundingGeometry.setHalfLengthWidthHeight( boundingVolumeDiagonal[0],
      boundingVolumeDiagonal[1], boundingVolumeDiagonal[2]);
```

Accessing Properties

The bounding volume now contains only its essential properties – the half width, height, and length of the lattice and the lattice offset. It no longer contains a position, a world bounding volume, a position or origin, or an orientation. Nor does it store the results of intersection tests, as these have now been moved to a separate intersector class, if not removed entirely. Later in this section, there are examples of how properties of the previous bounding volume were accessed, and how they are accessed now

Getting the Bounding Volume

The previous bounding volume class held two bounding volumes, a local bounding volume where local was not database coordinates, but lattice coordinates, and a transformed (database) bounding volume. These were both accessed in the same manner:

```
DtDcmRef worldBoundingVolume;
entityBoundingVolume.GetBoundingVolume(worldBoundingVolume);

DtDcmRef localBoundingVolume;
entityBoundingVolume.GetLocalBoundingVolume(localBoundingVolume);
```

The new bounding volume class is the lattice coordinate system bounding volume. So instead of presenting an interface for retrieving the entire bounding volume, it presents a more appropriate interface to the half-height, half-width, and half-length. Note that almost always, when developers accessed the bounding volume, they were actually trying to access the components of that bounding volume.

So to access the dimensions of the bounding volume, or to get the offset, call the appropriate member:

```
double length = entityBoundingVolume.length();
double halfWidth = entityBoundingVolume.halfWidth();
```

A more important change to the interface, for mutating the bounding volume, is documented in the next section.

Changing the *DtBoundingVolume* Class (Removal of the *mutate()* function):

The *mutate()* function was removed in favor of a simpler and more correct method for altering the structure of a bounding volume. It has been replaced by the *stretchForward()* and *stretchBackward()* functions. These functions alter the bounding volume in the expected manner.

For example, the collision detector used to mutate the collision bounding volume in order to stretch it out to detect potential collisions in front of the entity:

```
DtDcm entityLocalBV;
entity()->vrfState()->boundingVolume()->
    GetLocalBoundingVolume(entityLocalBV);

double oldRadiusInX = entityLocalBV [0][0];

double stoppingDistance = 1.0;
stoppingDistance = groundAutoController->
    calculateCurrentBrakingDistance();

double newRadiusInX = 0.0;
newRadiusInX = (stoppingDistance * stoppingDistanceFactor()) +
    oldRadiusInX;

entityLocalBV [0][0] = newRadiusInX;
entityLocalBV [1][1] += obstructionSensorDesc->lateralClearance();

DtVector bvolOffset;
bvolOffset[0] = newRadiusInX - oldRadiusInX;
bvolOffset[1] = 0.;
bvolOffset[2] = 0.;

myCollisionBV->Mutate(entityLocalBV, newRadiusInX, bvolOffset,
    bvolOffset[0]);
```

Now, the collision detector determines by what amounts it should stretch the bounding volume, and then does so. Note that it now has to reset the bounding volume to its initial state, or the stretch operations will be compounded.

```
// Make sure to reset the collision bounding volume to the default (actual
// entity local bounding volume) before we stretch it
const DtBoundingVolume& entityBvol = entity()->vrfState()->
vrfObjectGeometry()->localBoundingGeometry().boundingVolume();
*myCollisionBV = entityBvol;

// modify our local bounding volume in the x dimension by adding braking
// distance. The x dimension should be forward, so if we're ever not
// moving in the forward direction then this is incorrect. (such as for
// helicopters, submarines, etc.) Note that reverse is also OK, since it
// is still along the local (body) x axis.
double stoppingDistance = 1.0;
calculateStoppingDistance(*obstructionSensorDesc, stoppingDistance);

double lengthStretch = stoppingDistance * obstructionSensorDesc->
    stoppingDistanceFactor();
double widthStretch = obstructionSensorDesc->lateralClearance();

if(entity()->vrfState()->currentSpeed() >= 0)
{
    myCollisionBV->stretchForward( lengthStretch, widthStretch );
}
else
{
    myCollisionBV->stretchBackward( lengthStretch, widthStretch );
}
```

General Use of Bounding Volumes

Most uses of the bounding volume, other than for its dimensions, involve a bounding volume and its origin in local (database) coordinates. For many reasons, this data was removed from the *DtBoundingVolume* class. The following code constructs it, given a bounding volume, a transformation matrix, and a local location.

```
DtBoundingVolume bv;
DtVector bvPosition; // local position
DtDcm latticeToRef; // transformation from lattice to reference CS.

DtDcm worldBV;

// Create a diagonal matrix from the vector.
DtDcm tempMatrix;
tempMatrix[0][0] = bv.halfLength();
tempMatrix[1][1] = bv.halfWidth();
tempMatrix[2][2] = bv.halfLength();

// Rotate the world bounding volume in world coordinates.
DtDcmDcmMul(latticeToRef, tempMatrix, worldBV);

// And now calculate the reference origin
DtVector worldOrigin;
if(bv.isOffset())
{
    DtVector worldOffset;
    DtDcmVecMul(latticeToRef, bv.latticeOffset(), worldOffset);
    DtVecAdd(bvPosition, worldOffset, worldOrigin);
}
else
{
    worldOrigin = bvPosition;
}
```

The following example, taken from source code, converts the bounding volume from a lattice coordinate system to a transformed coordinate system, in this case the local coordinate system. To calculate an extent from an entity's bounding volume, the local (database) bounding volume must be calculated, and then used to expand an extent so that we have the smallest axis-aligned bounding box (what our *DtExtent* class is) that bounds the transformed bounding volume.

(This is the actual implementation of the function for expanding an extent to hold a bounding volume.)

```
void DtExpandExtentToIncludeBoundingVolume(DtExtent& extent,
    const DtBoundingVolume& boundingVolume, const DtDcm& worldOrientation,
    const DtPoint& worldLocation)
{
    DtVector vecWorldLocation(worldLocation.x(), worldLocation.y(),
        worldLocation.z());

    double halfLength = boundingVolume.halfLength();
    double halfWidth = boundingVolume.halfWidth();
    double halfHeight = boundingVolume.halfHeight();

    // First determine the local extent, in the lattice CS
    std::vector<DtVector> localExtentPoints(8);
```

```
// lower, back, left corner
localExtentPoints[0][0] -= halfLength;
localExtentPoints[0][1] -= halfWidth;
localExtentPoints[0][2] -= halfHeight;

// lower, back, right corner
localExtentPoints[1][0] -= halfLength;
localExtentPoints[1][1] += halfWidth;
localExtentPoints[1][2] -= halfHeight;

// lower, front, left corner
localExtentPoints[2][0] += halfLength;
localExtentPoints[2][1] -= halfWidth;
localExtentPoints[2][2] -= halfHeight;

// lower, front, right corner
localExtentPoints[3][0] += halfLength;
localExtentPoints[3][1] += halfWidth;
localExtentPoints[3][2] -= halfHeight;

// upper, back, left corner
localExtentPoints[4][0] -= halfLength;
localExtentPoints[4][1] -= halfWidth;
localExtentPoints[4][2] += halfHeight;

// upper, back, right corner
localExtentPoints[5][0] -= halfLength;
localExtentPoints[5][1] += halfWidth;
localExtentPoints[5][2] += halfHeight;

// upper, front, left corner
localExtentPoints[6][0] += halfLength;
localExtentPoints[6][1] -= halfWidth;
localExtentPoints[6][2] += halfHeight;

// upper, front, right corner
localExtentPoints[7][0] += halfLength;
localExtentPoints[7][1] += halfWidth;
localExtentPoints[7][2] += halfHeight;

if (boundingVolume.isOffset())
{
    // Make sure to account for the lattice offset if it exists
    for (int pointIndex = 0; pointIndex < 8; ++pointIndex)
    {
        DtVecAdd(localExtentPoints[pointIndex],
            boundingVolume.latticeOffset(), localExtentPoints[pointIndex]);
    }
}

std::vector<DtVector> worldExtentPoints(8);
for (int pointIndex = 0; pointIndex < 8; ++pointIndex)
{
    DtDcmVecMul(worldOrientation, localExtentPoints[pointIndex],
        worldExtentPoints[pointIndex]);

    DtVecAdd(worldExtentPoints[pointIndex], vecWorldLocation,
        worldExtentPoints[pointIndex]);

    extent.expandToInclude(worldExtentPoints[pointIndex]);
}
```

```

    }

    return;
};

```

For more information, see the *DtBoundingVolume* and *DtBoundingVolumeIntersector* class documentation, as they contain extensive documentation of these concepts.

9.8.11. Changes to the *DtCgf* Class

The *DtCgf* class has been modified in the following ways:

- ♦ Access to the terrain database member has been restricted to accessor functions so that you can easily override the type of *DtTerrainDatabase* used by the application. You now set or get the terrain database through the accessors (the member itself is now private).
- ♦ You can now create a derived *DtTerrainDatabase* and replace the one shipped with VR-Forces. The *DtCgf* must now keep track of the *vrfCreator* object. *DtCgf* does not maintain ownership of user specified *vrfCreator* objects, but will do so for instances it creates. This *vrfCreator* is used to create the terrain database. Thus by registering a new create function for a derived *DtTerrainDatabase* with the *vrfCreator* and then by supplying that creator as a parameter to the *DtCgf::init()* function, you can specify that your derived *DtTerrainDatabase* be used.

The *DtVrfCreator* and related classes have been updated with the ability to register or use creation functions for the *DtTerrainDatabases* and to create instances of them.

- ♦ The *DtFireManager* and *DtDetonationManager* have been added. They are responsible for processing fire and detonation interactions, and for working with the *DtDamageAdjudicationActuator* and the *DtUnderFireDeterminationSensor* for adjudication damage and determining under-fire status. The *DtCgf* class owns these objects and is responsible for their construction and destruction. See “[Centralized Fire and Detonation Processing](#),” on page 9-22.
- ♦ New functions for logging frame rate statistics have been added. These are purely additive changes, and do not affect backwards compatibility at all.
- ♦ The copy and assignment operators were made explicitly protected and not implemented to guarantee that the compiler did not generate them. (This was assumed true and intended before.)

Explicit Hiding of Functions

Many classes in the API were hiding functions from the classes they derived from. In most cases, this was intentional. However, to ensure that developers know the hiding is intentional, and to remove any potential compiler (or other tool) warnings, many of the hidden functions were explicitly hidden by declaring them as `in use, but private`. For example, in the *DtVrfObjectManager* class, the following was added:

```

private:
    // Intentionally hiding these functions

```

```
// They have been replaced by new declarations/definitions in this
// class.
using DtRwList::add;
using DtRwList::remove;
```

***DtSpecificationAttribute* Changed to Use Symbol Strings**

The *DtSpecificationAttribute* class has been updated to contain a symbol string instead of an actual string. This significantly reduces the amount of memory needed to store large numbers of specification attributes in the specifications. Most specifications contain the same attributes, and so this reduction is significant for terrain files with large amounts of vector data. You should not have to do anything to update your code. Additionally, the label member has been made private, but with public accessors so that derived classes can easily override the functionality to store the label another way. For more information, see [“Symbol Strings,”](#) on page 9-36.

9.8.12. Spatial Organization of Objects

The *DtVrfObjectManager* class now contains a *DtSpatialVrfObjectManager*, which it uses to spatially organize specified *DtVrfObjects*. A filter function predicate is used to determine which *DtVrfObjects* are spatially sorted, in order to allow you to easily change the sorting in a derived class. Additionally, the creation, initialization, and use of the *DtSpatialVrfObjectManager* class is all performed in the appropriate functions, allowing for user modification. For a complete description of spatial subdivisions, see [“Spatial Subdivisions,”](#) on page 9-31.

A simple algorithm is used to determine the spatial organization configuration (number of subdivisions in each dimension). Also, all *DtVrfObjects* are stored in the Spatial Manager, including routes, waypoints, and so on.

To access the spatially sorted *vrfObjects*, the *getVrfObjects()* functions are provided to get all *vrfObjects* that may intersect the specified primitive (currently only *DtChord* or *DtExtent*). The resulting set contains all objects that may intersect the chord or extent. Not all are guaranteed to do so, but the set will not be missing any that actually do. If the exact intersection is required, as for line-of-sight testing in the *vrfObjectStateRepository*, then further refinement is required.

Additionally, the functions, *DtRwList::add* and *DtRwList::remove*, that were being hidden implicitly have been made hidden explicitly. This does not change the API, but makes it clear that the functions are being hidden on purpose.

For more details on how to override the spatial organization behavior in both this class and of the vector obstructions, see the appropriate sections in the manual.

Changes to *DtVrfObjectStateRepository*

The *DtVrfObjectStateRepository* now uses the spatially managed *vrfObjects* and vector obstructions when determining line-of-sight between entities. The existing function, *intersectsEntity()*, now calls a new function, *intersectEntities()*, to see if the specified chord intersects any entities other than the entity being tested for LOS. As a note, the same predicate used for determining if an entity is to be tested for LOS is used to screen out entities from the spatially sorted intersection results.

Changes to *DtTerrainDatabase*

The *DtTerrainDatabase* class now uses a *DtSpatialSubdivision* for spatially organizing the terrain, instead of a *DtSpatialIndex*. The *DtSpatialIndex* remains in the library, and is used for verification purposes against the *DtSpatialSubdivision*, but it is no longer supported for other use.

The accessor functions for the terrain database should change slightly. For example, to request the spatial subdivision, instead of saying:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = terrainDb.spatialIndex();
```

the following is used:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = terrainDb.spatialSubdivision();
```

The reason the name was changed explicitly, instead of just changing what the spatial index was, was to make sure that there was no confusion between the old uniform spatial container for the terrain, and the new generic uniform spatial container, which provides a minimal interface. See the details of the *DtSpatialSubdivision* for more information.

Additionally, the following were all updated:

- ♦ The version of the GDB format was incremented, specifically to account for the addition of the spatial subdivision.
- ♦ *transformCoordinateSystem()* now takes an optional parameter that allows the existing coordinate system to be retransformed to itself. This is useful for reprojecting a coordinate system to a new origin.
- ♦ New functions for finding the ground height and the intersection at that location have been provided: See *terrainHeightAndIntersection()* in the class documentation.
- ♦ Support for the Flat Earth coordinate system projection has been added, as well as functions for transforming to it and from it. Functions for transforming from UTM to UTM have also been added, which are to be used to reposition an existing coordinate system.
- ♦ The memory manager member has been made mutable, as it is modified by const accessor functions.

9.8.13. Spatial Subdivisions

The *DtSpatialSubdivision* class represents a uniform spatial container. The container is divided up into a lattice of cells, with resolution specified at creation. Each cell is a container of the objects in it. Its purpose is to efficiently store and search objects spatially. In this regard it is the same as the original *DtSpatialIndex* class. However, spatial subdivisions are more flexible and extensible.

The *DtSpatialSubdivision* is a templated container, similar to the `std::list` or `std::vector` classes. As such, it can hold any specified item. The intended use is to store elements that have “spatial” properties – that is elements that have an actual volume, position, orientation, and so on. Because of this flexibility, not only is the terrain spatially sorted in VR-Forces 3.8, but so are the *DtVrfObjects* and the *DtVectorObstructions*, resulting in a significant improvement in performance when simulating large numbers of entities.

The responsibility for and intelligence necessary to perform operations on the spatial subdivision, such as intersection with primitives, insertion of elements into, and so on, have been separated out into specific classes for the distinct operations. This greatly facilitates extending existing capabilities and adding new ones. As a result, the spatial subdivision functionality is spread out across an entire family of classes, as described in this section.

Spatial Subdivision Container

The *DtSpatialSubdivision* is the container class. It is defined in *spatialSubdivision.h*, in the geometry library.

Spatial Subdivision Intersectors

The spatial subdivision no longer contains the ability to intersect geometric primitives and other objects with itself. The ability to do that has been moved into a set of intersector classes. These intersector classes know how to intersect a primitive and a spatial subdivision, and apply a specified operation to the results of the intersection through a functor interface.

The functor interface allows for tremendous flexibility in the type and complexity of operations that are performed on the spatial subdivision. By changing the functor supplied to the `intersect()` member function, different operations can be performed, such as inserting elements into the spatial subdivision, gathering elements of specific characteristics from the spatial subdivision, or simply accumulating all cells in the spatial subdivision that are part of the intersection. The following are examples of inter-sectors:

- ♦ *DtSsChordIntersector* (*ssChordIntersector.h*) – This class intersects a specified chord and spatial subdivision, applying a functor to the intersection results.
- ♦ *DtSsExtentIntersector* (*ssExtentIntersector.h*) – This class intersects a specified extent and spatial subdivision, applying a functor to the intersection results. Note that this class also provides a function for determining the min/max intersected cell indices, which can sometimes be more useful than the actual full intersection call.

Spatial Subdivision Functors

The spatial subdivision functors are simple classes that are used by the spatial subdivision intersectors to perform different operations on the results of intersection tests. These operations include insertion of elements into spatial subdivision cells (*DtSsInsertionFunctor*), accumulation of spatial subdivision cells for later analysis (*DtSsAccumulationFunctor*), among others.

While simple, these functors differ from standard functors in the following important ways:

- The functors are passed by **reference**, not by value. They do not have to maintain state, but they are expected to.
- The functors return a Boolean to indicate whether or not they are done processing inputs, and **not** whether or not an error occurred. This is to allow for efficient processing of intersection results. For instance if a functor is looking for any intersection of a chord and a polygon, it can stop processing after the first intersection is found, simply by returning true. To determine if an error occurred, the functor **must** either maintain internal error state or use exceptions.

The following are examples of functors:

- *DtSsFunctor* (*ssFunctor.h*) – This is the base class for all spatial subdivision functors. It does not perform any operation.
- *DtSsAccumulationFunctor* (*ssAccumulationFunctor.h*) – *DtSsAccumulationFunctor* contains a container, which it uses to store elements passed in to its function operator (see the function definition for more details). Thus it can be used to accumulate the set of cells comprising the intersection of a primitive with the spatial subdivision.
- *DtSsInsertionFunctor* (*ssInsertionFunctor.h*) – *DtSsInsertionFunctor* is used to insert one object into another, for instance in inserting elements into the spatial subdivision cells.

Spatial Subdivision Inserters

The spatial subdivision inserter classes are utility classes that combine the appropriate intersector with an insertion functor, in order to facilitate inserting of elements of the appropriate type into a spatial subdivision. *DtSsGdbNodeInserter* (*ssGdbNodeInserter.h* and *ssGdbNodeInserter.cxx*) is an example of an inserter. It inserts *DtGdbNodes* into a spatial subdivision. Generally speaking, it uses an extent intersector to find the cells which the *DtGdbNode* intersects, and then an insertion functor to insert the *DtGdbNode* into those cells.

Migrating from *DtSpatialIndex* to *DtSpatialSubdivision*

This section explains how to migrate existing code from using the *DtSpatialIndex* to using the new *DtSpatialSubdivision*. For more information, please consult the class documentation.

Creating the Container

Creation has not really changed, except for the specification of the template parameter.

Previously:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = new DtSpatialIndex(params);
```

Now:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = new DtSpatialSubdivision<DtGdbNode*>(params);
```

Notes

- ♦ To change the type of contained element, change the template parameter – `DtGdbnode*`
- ♦ The parameters to the constructor functions have not changed.

Accessing the Container from the Terrain Database

The spatial subdivision is accessed in the same way as the spatial index, except that the function names have been changed to be `spatialSubdivision`.

Previously:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = terrainDatabase->spatialIndex();
```

Now:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = terrainDatabase->spatialSubdivision();
```

Inserting Elements into the Container

Previously, to insert a node into the spatial index, the `addNode()` function was called. For example, when the terrain database created the spatial index for the terrain, it used the following code, where `mySpatialIndex` was the terrain database's spatial index and `myTerrain` was the terrain database's root terrain *DtGdbNode**:

```
mySpatialIndex->addNode(myTerrain);
```

However, using the new spatial subdivision, an inserter class must be used, because the spatial subdivision no longer knows how to insert other objects into itself.

The following example show how the terrain database now creates the spatial subdivision for the terrain polygons. The `terrain()` function returns the root terrain polygon node (*DtGdbNode**).

```
DtSsGdbNodeInserter terrainInserter;
if (!terrainInserter.insert(terrain(), *myTerrainSpatialSubdivision))
```

```
{
    error handling code...
}
```

Finding Terrain Intersections with the Terrain's Spatial Container

Previously, the spatial index provided a set of intersect functions that matched the intersect functions of the terrain database. These all took chords or bundles of chords as parameters, as follows:

```
spatialIndex->intersect(chord, intersectionPoint, intersectionTime);
```

Now, there is a helper class for finding terrain intersections, called *DtTerrainIntersector* (*terrainIntersector.h*). The terrain intersector contains all the knowledge of the group of different terrain intersection functions the spatial index supported, and in turn calls the appropriate intersectors, functors, and performs the appropriate analysis on the results. To find terrain intersections, do the following:

```
DtTerrainIntersector terrainIntersector;
terrainIntersector.intersect(chord, spatialSubdivision,
    intersectionPoint, intersectionTime);
```

Intersecting *DtChords* and *DtExtents* Directly with the Container

Previously, there was no way to explicitly intersect an extent with the spatial index, from a mathematical point of view, and get the resulting cells that made up the intersection - the logic was locked inside the spatial index class! Now the *DtSsChordIntersector* class (*ssChordIntersector.h*) contains all the intelligence for intersecting a *DtChord* and a spatial subdivision. To intersect a chord with an spatial subdivision, create an instance of a *DtSsChordIntersector*, and call its intersect function with the desired chord, spatial subdivision, and functor. See the class documentation for more information.

The following code is an abridged example of how an accumulation functor is used and then analyzed (some data has been omitted for brevity):

```
DtSsChordIntersector<DtGdbNode*> chordIntersector;
DtSsAccumulationFunctor<DtChord,
    DtSpatialSubdivision<DtGdbNode*>::DtSpatialSubCellType>
    accumulationFunctor;

if (chordIntersector.intersect(chord, spatialSubdivision,
    accumulationFunctor))
{
    // walk the accumulated cells, and analyze them appropriately
    DtAccumulationContainer::iterator cellIter =
        accumulationFunctor.container().begin();
    DtAccumulationContainer::iterator endCellIter
        accumulationFunctor.container().end();
    for (; cellIter != endCellIter; ++cellIter)
    {
        DtSpatialSubCellType* cell = *cellIter;
        DtSpatialSubCellType::iterator cellContentsIter = cell->begin();
        DtSpatialSubCellType::iterator endCellContentsIter = cell->end();

        for (; cellContentsIter != endCellContentsIter; ++ cellContentsIter)
        {
```

```

        // PERFORM ANALYSIS
    }
}

```

Intersecting *DtExtents* with the Container

As with *DtChords*, previously there was no way to explicitly intersect an extent mathematically with the spatial index, and get the set of cells that comprised the intersection - the logic was locked inside the spatial index class. Now, the *DtSsExtentIntersector* (*ssExtentIntersector.h*) contains all the intelligence for intersecting a *DtExtent* and a spatial subdivision. To intersect an extent with an spatial subdivision, create an instance of an *DtSsExtentIntersector* and call its intersect function with the desired extent, spatial subdivision, and functor. See the class documentation for more information.

i

Extents are the only non-chord primitive for which intersectorors are provided. While other intersectorors exist for *DtVrfObjects*, *DtVectorObstructions*, and *DtGdbNodes*, all create extents from the base objects that are then used to intersect the spatial subdivision.

```

DtSsExtentIntersector<DtVectorObstruction*> extentIntersector;
if (extentIntersector.intersect(vectorObstruction, spatialSubdivision,
    functor))
{
    // do whatever you want to do when there is an intersection
}
else
{
    // do whatever you want to do when there isn't an intersection
}

```

Extending the *DtSpatialSubdivision* Classes

Because the spatial subdivision is a templated container, creating a spatial subdivision of a new type of element is as easy as altering the declaration.

For example, the following creates a spatial subdivision of pointers to *DtGdbNodes*:

```
new DtSpatialSubdivision<DtGdbNode*> gdbNodeSpatialSub;
```

The following creates a spatial subdivision of pointers to *DtVectorObstructions*:

```
new DtSpatialSubdivision<DtVectorObstruction*>
vectorObstructionSpatialSub;
```

Creating A New Type of *DtSsfunctor*

Derive the new functor from the *DtSsFunctor* class. To use this functor, pass it as a parameter to an existing spatial subdivision intersector or other utility class, or to a newly created spatial subdivision utility class.



Be sure that the new functor adheres to the interface of the *DtSsFunctor* classes.

Creating A New Intersector

Creating new intersector classes is probably one of the most useful extensions to the spatial subdivision classes. For instance, an intersector class for spheres or cylinders or other implicit surfaces would enable efficient selection of all elements within a specified radius of a location (or line).

To create a new intersector class, there is no need to derive from any parent class. The intersector class merely needs to know how to intersect the specified primitive with the spatial subdivision and to allow user-specified processing of the results.

So, to create a sphere intersector you could test every cell for inclusion with the specified sphere, using the properties of the spatial subdivision (cell deltas, location, and so on) to calculate the cells to test, and to pass these cells to the user-specified functor. This would be useful when selecting all cells within a blast radius. To take it a step further, a new functor could also be derived that would know how to intersect all the elements of a cell, such as *DtGdbNodes*, with the specified primitive. Used together, the new sphere intersector and sphere intersection functor could be used to select all polygons within the blast radius of a detonation.

9.8.14. Symbol Strings

The *DtSymbolString* class, and related classes, provide a way to treat strings as symbols instead of as ordinary strings. When a symbol string is created from a string, the string is stored inside of a table of strings and the symbol string is given a key with which to get the associated string from the table. As a result, unique strings are stored only once. This can result in significant memory savings if many instances of the same string were formerly stored in the system. Additionally, because each symbol string references its string by a key, string comparison becomes key comparison (much more efficient). In general, the symbol string classes are used in the reader/writer classes and the *DtSpecification* classes. You should not have to update code to use the *DtSymbolStrings*, other than to provide the appropriate functions if any classes were derived from reader/writer classes implementing member functions using *DtSymbolStrings*.

The symbol string classes and header files, which are in *gdb.lib*, are:

- ♦ *DtSymbolStore* (*symbolStore.h*) – The repository of unique symbols. Used, as a singleton, by the symbol string class.
- ♦ *DtSymbolString* (*symStr.h*) – The core symbol string class.
- ♦ *DtCStringHashKey* (*cstringHashKey.h*) – A string hash key based upon symbol strings, instead of full strings.

To maintain backwards compatibility, you do not need to make any changes to your object parameters database files. To use the new classes, update the object parameters database files to use the damage-adjudication-actuator and under-fire-determination-sensor.

9.8.15. New Classes for Managing Aggregates

The following sections describe new classes that manage aggregate behaviors.

New Aggregate State Repository Class

DtVrfAggregateStateRepository (*vrfAggregateSR.h*) is a state repository class for all aggregate entities. It maintains the current type of formation, and the current state of the formation (which subordinate is in which formation position). All aggregate entities are configured with this type of state repository.

New Aggregate Process State Repository Classes

- ♦ *DtAggregateMoveAlongPSR* (*aggregateMoveAlongPSR.h*) contains the state associated with movement along a route task. It is used by the *DtAggregateMoveAlongController* class.
- ♦ *DtAggregateMoveToPSR* (*aggregateMoveToPSR.h*) contains the state associated with a move to task. It is used by the *DtAggregateMoveToController* class.
- ♦ *DtAggregatePatrolAlongPSR* (*aggregatePatrolAlongPSR.h*) contains the state associated with a patrol along route task. It is used by the *DtAggregatePatrolAlongController* class.
- ♦ *DtAggregatePatrolBetweenPSR* (*aggregatePatrolBetweenPSR.h*) holds state associated with a patrol between task. It is used by the *DtAggregatePatrolBetweenController* class.

New Aggregate Controller Component Classes:

- ♦ *DtAggregateFormationController* (*aggregateFormationController.h*) manages the state associated with an aggregate entity's formation. This includes determining who is eligible to participate in the formation and who should be positioned relative to whom to form the formation. It also handles all *DtSetDataRequests* that affect formation, including *DtSetLocationRequests*, *DtSetHeadingRequests*, and *DtSnapIntoFormationRequests*. It sets the state in the aggregate entity's *DtFormationState* object (owned by the *DtVrfAggregateStateRepository* class).
- ♦ *DtAggregateMoveAlongController* (*aggregateMoveAlongController.h*) models movement of an aggregate along a given route. It registers for *DtMoveAlongTask* messages. Execution of the behavior is broken down into two phases. First, it issues a *DtMoveIntoFormation* subtask to itself (handled by the *DtMoveIntoFormationController*) to move the aggregate into position at the start of the route, in the given formation, and oriented in the correct direction. When this subtask is complete, it generates a separate 'working route' for each subordinate to follow, and sends a *DtMoveAlongTask* to each subordinate with its route. The 'working routes' generated are created at the offset relative to the original route necessary to maintain the proper lateral position of the subordinate in formation. This controller periodically monitors how well each subordinate is maintaining its position in formation and issues Set Speed Requests to speed up or slow down a given subordinate to try to maintain the correct front-to-back shape of the formation. The task is complete when all subordinates have sent a *DtTaskComplete* report message to the aggregate indicating they have reached the end of their respective 'working routes'.
- ♦ *DtAggregateMoveIntoFormationController* (*aggregateMoveIntoFormationController.h*) models movement of an aggregate into formation at a given position and heading. It registers for *DtMoveIntoFormationTask* messages.
- ♦ *DtAggregateMoveToController* (*aggregateMoveToController.h*) models movement of an aggregate to a given waypoint. It registers for *DtMoveToTask* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself.
- ♦ *DtAggregateMoveToLocationController* (*aggregateMoveToLocationController.h*) models movement of an aggregate to a given waypoint. It registers for *DtMoveToLocation* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself.
- ♦ *DtAggregatePatrolAlongController* (*aggregatePatrolAlongController.h*) models patrol along a route behavior. It registers for *DtPatrolRoute* messages. Its behavior is implemented by alternating between two subtasks: *DtMoveAlongTask* and *DtReverseDirectionTask*. When it first receives a *DtPatrolAlongTask*, it issues a *DtMoveAlongTask* subtask to itself. When this task is complete (it receives a *DtTaskComplete* report from itself), it issues a *DtMoveAlongTask* to itself in the reverse direction. It repeats the process indefinitely, alternating directions.
- ♦ *DtAggregatePatrolBetweenController* (*aggregatePatrolBetweenController.h*) models patrol between two waypoints behavior. It registers for *DtPatrolBetween* messages. Its behavior is implemented by sending *DtMoveTo* subtasks to itself, alternating between the two waypoints indefinitely.

- ♦ *DtAggregateTurnToHeadingController* (*aggregateTurnToHeadingController.h*) models behavior of turning to a given heading. It registers for *DtTurnToHeading* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself at its current location, current formation, and new heading.

New Task Class

- ♦ *DtMoveIntoFormationTask* (*moveIntoFormationTask.h*) tasks an aggregate to move into a given formation, at a given location and heading.

New Set Data Request Classes

- ♦ *DtSnapIntoFormationRequest* (*snapIntoFormationRequest.h*) requests that an aggregate instantly reposition and reorient all of its subordinates into a given formation, at a given position and heading.
- ♦ *DtSetFormationTypeSetDataRequestType* (*setFormationType.h*) requests that an aggregate set its formation type to a given formation enumeration. It does not change the relative positioning of subordinates in formation.

Sending Task Messages

Aggregate controller components are now responsible for sending clear task messages to the subordinates to which they assign tasks. This was formerly done by the *DtPseudoAggregateTaskManager* (now obsolete). This was moved to aggregate controllers because that is where the logic for issuing tasks to subordinates resides.

Obsolete Classes

The following classes have been removed from the VR-Forces API:

- ♦ *DtPseudoAggregateFormation* (*psdAggFrmCtrl.h*)
- ♦ *DtPseudoAggregateMoveController* (*psdAggMvCtrl.h*).
- ♦ *DtPseudoAggregateTaskManager* is obsolete. Aggregates should now be configured with the same *DtLocalTaskManager* class as individual entities. *DtPseudoAggregateTaskManager* has been typedef'd to *DtLocalTaskManager* for backward's compatibility.

9.8.16. Ability to Extend State Repositories at the Base Class Level

You can now extend the base state repository class *DtVrfObjectStateRepository* (*vrfObjSR.h*), as an alternative to, or in addition to extending leaf-level classes in the *DtVrfObjectStateRepository* hierarchy. The base state repository class, *DtVrfObjectStateRepository* now has a *DtVrfStateRepositoryUserExtension* (*vrfSRUserExt.h*) that you can extend for all types of entities and objects.

For example, suppose you want to extend all types of entities and objects in VR-Forces (fixed-wing, rotary-wing, ground, and so on) with a new kind of identifier. Rather than extending each type of state repository to include the new identifier, you just need to create a single derived kind of *DtVrfStateRepositoryUserExtension* class. This allows you to write the extension once, and configure it as part of a variety of different kinds of entities, control objects, or overlay objects. The alternative to this approach would be to subclass each kind of state repository separately, extending each class in exactly the same way.

Like the *DtVrfObjectStateRepository* classes, classes derived from *DtVrfStateRepositoryUserExtension* are readable and writeable. Any *DtReaderWriter* data members you add to your derived *DtVrfStateRepositoryUserExtension* will be saved to an order of battle file along with the rest of an object's *DtVrfObjectStateRepository* data.

For complete details, see [“Extending State Repositories at the Base Class Level.”](#), in *VR-Forces Developer's Guide*.

9.8.17. Terrain API Changes

This section summarizes changes to the Terrain API.

GeometryNative Files Removed

The *geometryNative* project and files that were deprecated in the 3.6 release have finally been removed. Any remaining references to them should be changed to use the appropriate files in the *Geometry* project.

***DtSpecificationAttribute* Refactored**

The *DtSpecificationAttribute* class has been updated to contain a symbol string instead of an actual string. This significantly reduces the amount of memory needed to store large numbers of specifications attributes in the specifications. Most specifications contain the same attributes, and so this reduction is non-trivial for terrain files with large amounts of vector data. You should not have to do anything to update your code. Additionally, the *label* member has been made private, but with public accessors so that derived classes can easily override the functionality to store the label another way.

New GDB File Version

The GDB file format has been incremented to account for the spatial subdivision changes. VR-Forces can read GDB files using earlier formats, but previous versions of the VR-Forces cannot read GDB files created with the release 2.83.10 *tdbtool*.



The new GDB version no longer explicitly stores the entire spatial subdivision in the file. It stores only the configuration, so that it can be recreated every time the file is imported. This only marginally affects the performance of importing GDB files, and drastically reduces versioning issues when reading GDB files.

9.8.18. Miscellaneous VR-Forces API Changes

The following changes were made to the VR-Forces API:

- ♦ Added ability to easily derive and use new *DtTerrainDatabase* classes.
- ♦ *DtNetIfModifyVrfObject* did not have myAppearance in the structure nor was it set over the network.
- ♦ *DtSymbolDrawer* in the *DtSymbol* object is now a pointer rather than a reference.
- ♦ The code that determines if an entity is considered to be under fire has been moved from *DtVrfObjectStateRepository* to damage actuators and the under fire detection sensor. The primary reason for this is that state repositories are designed to store state, not determine what the state should be. The determineUnderFire() member functions still have the same signatures and can be overridden from the damage actuators exactly as they could be from the *DtVrfObjectStateRepository*.
- ♦ The under fire parameters have been moved from *DtVrfObjectParameters* to *DtDamageComponentDescriptor*, however for backwards compatibility they can still be specified in their original location in the object parameters database. In that case they will override the values specified in the entry for the damage actuator.
- ♦ The *DtSimObjectStateRepository::tick()* member function has been removed because it was replaced by *DtVrfObjectStateRepository::tick()* in a previous version of VR-Forces. It has been removed as a clean-up measure; *DtVrfObject* ticks a *DtVrfObjectStateRepository*, not a *DtSimObjectStateRepository*.
- ♦ The *DtSimpleStats* template class has been added. (The file *simpleStats.h* has been added to the geometry library.) It is for use in manual profiling. This is a simple class for maintaining statistics such as average, min, max, sum and count. The class intentionally does not provide support for more complex computations, nor does it maintain a list of N number of items added. This is to ensure that its performance is O(1) and its usage cost low enough for use in the most performance critical areas of code.
- ♦ *DtOutputLogger* is a simple class for logging information to a file. To be used, the caller must specify a logging level as well as a global logging level. If the threshold is less than the globalThreshold, then the output logger will output the specified information to the associated file, (as long as the file was created successfully). See the header file *outputLogger.h* for more information. It was added to the vrfUtil library.
- ♦ *DtVrfObjectManager* has a new member function, createSubObjectFromParameterEntry(). It provides a place for you to extend during the process of creating an aggregate with subordinates.
- ♦ In the class *DTVrfObjectParameters*, the member variables mySubordinateObjects and myAttachedObjects are now dynamically allocated, so you can override their creation and create derived types.
- ♦ In *DtGroundAutomotiveController* (*gndAutoCntlr.h*), the functions accelerationSoilFactor() and maxSpeedSoilFactor() are obsolete and will be removed in future versions of VR-Forces. They both return a value of 1.0, so neither incorporates effects due to soil types. The functions are being removed because the effects of soil

factors on acceleration and speed are already taken into account in the *DtAutomotiveActuator*. By checking for these effects in both the controllers and the actuator, we were taking soil factors into account twice as much as we should have.

- ♦ To prevent the *DtWeaponController* from firing a pre-loaded munition at an entity that it does not have an ammoselect table entry for, the tick function now calls `loadedAmmoValidForTarget()`, which verifies that the ammo is loaded and appropriate for the target type before it calls `fire()`.
- ♦ The `myCurrentMunition` and `myCurrentFuze` member variables were removed from *DtBaseWeaponController* because these values were moved to the *DtVrfWeaponPSR*. The values can now be accessed by calling `weaponProcessState()->munitionLoader().currentMunition()` or `weaponProcessState()->munitionLoader().currentFuze()`.
- ♦ The class interface to *DtReaderWriter* has changed to allow an explicit search path to be used when reading and writing files. Previously, all relative filenames specified within a *DtReaderWriter* file were assigned to be relative to the application path. See *VR-Forces Developer's Guide* and *rdrWrtr.h* for details.

Ability to Check for VR-Forces Licenses at Runtime

VR-Forces can check the availability of VR-Forces licenses at runtime. This lets you write code to handle a situation in which an expected license is unavailable for some reason. For example, you might want to use the opportunity to pop up a warning to alert the user. You can check for the availability of back-end, remote control API, or GUI licenses. For details, see *vrfCheckLicense.h*.

The following example shows how to embed a VR-Forces back-end in a manned simulator. Before trying to create an instance of the top-level back-end class (*DtCgf*), it checks to see if a back-end license is available. If not, it prints a warning message.

```
#include <vrfCheckLicense.h>

...

if (DtHaveVrfSimLicense() == true)
{
    DtCgf* cgf = new DtCgf(addr);
    cgf->init(exerciseConn);

    ...
}
else
{
    DtWarn << "Error checking out a VR-Forces back-end license."
    << std::endl;
    ...
}
```

getProcessSR() Member Functions Renamed

The `getProcessSR()` member functions were renamed to `lookupAndCacheProcessSR()` and the comments were improved to clarify that the member function is not an accessor (it returns void) and that it is used to update a class's pointer to its process state repository. In previous versions, some `getProcessSR()` member functions were public and some were protected. Now all `lookupAndCacheProcessSR()` are protected to indicate that this member function is to be used only to update these internal pointers. The affected classes are:

- ♦ *DtBallisticGunController*
- ♦ *DtTurretActuatorComponent*
- ♦ *DtSurfaceEntityPilotController*
- ♦ *DtRotaryWingPilotController*
- ♦ *DtPseudoAggregateSetController*
- ♦ *DtPseudoAggregateFormationController*
- ♦ *DtMissileControllerComponent*
- ♦ *DtIndirectFireWeaponController*
- ♦ *DtGroundAutoControllerComponent*
- ♦ *DtBaseWeaponController*
- ♦ *DtBallisticGunController*
- ♦ *DtFixedWingPilotController*.

DtCollisionDetector

Overall, the collision detector class was refactored to improve the interface, and make it much easier to override key aspects of the functionality. (For example, in order to change the behavior to return a list of obstructions, rather than the most imminent obstruction, or to institute a list of obstructions to ignore.) To this end, the following was done:

- ♦ The ability to remove object and vector obstructions IDs was added.
- ♦ A better, cleaner iterator type interface was added to protect against future changes in the API, as well as to make the interface clearer and easier to use.
- ♦ Typedefs have been added at the top of the class in order to further abstract the internal representation of the class away from the interface.
- ♦ The class now guarantees that only single instances of obstructions will be returned in the results. This motivated a change from using `std::lists` to `std::sets`, but that is an implementation detail. It is this type of change that the typedefs will help ameliorate.
- ♦ The list of obstructions are no longer cached internally. They are selected every collision query from the appropriate spatial manager.

The add and remove ID functions no longer take a container of IDs to add/remove, but rather a single ID. This is to remove the knowledge of what type of container the items are stored in the *DtObstructionSensor* from the *DtCollisionDetector*. That way, customer wishing to replace either will find it easier to do so.

Also, because the container types have changed as mentioned above, it is best to change any references to them in the code with typedef'd types at the top of the file. For example, in 3.7.1 API, you would have said the following:

```
std::set<DtVrfObject*> objectObstructions;
if (!collisionDetector.getObjectObstructions(objectObstructions))
{
    ...
}
```

Now, just say the following:

```
DtObjectObstructionContainer objectObstructions;
if (!collisionDetector.getObjectObstructions(objectObstructions))
{
    ...
}
```

DtObstructionSensor

The *DtObstructionSensor* has been updated to make it much easier to replace the collision detector that is used to detect collisions. While not backwards compatible, this change makes it *much* easier to derive a new *DtCollisionDetector*, and specify that the *DtObstructionSensor* use it.

Specifically, functions for creating the collision detector, initializing the obstruction IDs to use from a descriptor and for calculating the stopping distance (part of the obstruction avoidance algorithm) also from a descriptor have been added.

See the header file for more details about the new functions:

```
virtual bool createCollisionDetector();
virtual bool initializeObstructionIDs(
    const DtObstructionSensorDescriptor& obstructionSensorDesc);
virtual bool calculateStoppingDistance(
    const DtObstructionSensorDescriptor& obstructionSensorDesc,
    double& stoppingDistance);
```

Setting Appearance Attributes

DtVrfRemoteController has a new member function, `setAppearance()`, that lets you set appearance attributes for a particular entity. A new message, *DtSetAppearanceRequest*, has been added to send this value to the back-end.

The member function `processSetAppearance()` in the *DtLocalEntitySetController* can be overridden in order to do special processing on appearance attribute values.

Task Manager/Task Controller Changes

DtPseudoAggregateTaskManager is obsolete. Aggregates should now be configured with the same *DtLocalTaskManager* class as individual entities.

Aggregate controller components are now responsible for sending clear task messages to the subordinates to which they assign tasks. This was formerly done by the *DtPseudoAggregateTaskManager* (now obsolete). This was moved to aggregate controllers because that is where the logic for issuing tasks to subordinates resides.

When you register for a task message in a *DtTaskController*, make sure you call `startTask()` before you set up any of the new task. The first thing `startTask()` does is clean out the old task in whatever controller was executing a top-level task at the time the new task came in (retask). This retask could involve the component who has received the new task, so you do not want to initialize anything before `startTask()` is invoked that will end up being cleaned out by `clearTask()`.

Components derived from *DtTaskControllerComponent* will have their `clearTask()` member function called when either of the following is true:

- ♦ Skip task has been called, and task controller is currently executing a top-level task
- ♦ An entity has been retasked.

Change to *DtFltReader*

The `xlateGroupChildrenIfAny()` function now takes a reference to a pointer to a *DtGroup*, instead of an actual *DtGroup* pointer, so that it can allocate the group when and if a child node is translated. This avoids creation of *DtGroup* nodes that have no children.

Changes to *utils.h* and *utils.cxx*

The file I/O and general utility functions in *utils.h* have been updated to be compliant with our current coding specification. Additionally, several macros have been removed in favor of direct function calls that accomplish the same end. The following macros were removed:

- ♦ `DEBUGMACRO`
- ♦ `PRINTINFO`
- ♦ `PRINTWARN`

The VR-Link functions, *DtWarn*, *DtDebug*, and *DtInfo* should be used instead. Additionally, the function `trap_to_debugger` was removed. Instead, `assert()`, or some other suitable debugging function, should be used in its place.

Addition of *mathUtilities.h*

A new file of math related utility functions and classes has been added to the geometry project. This file will eventually be moved, but for now, it resides in the geometry project. The file contains math related utility functions, such as floating point comparison operations.

Deprecated Classes

Many areas of the code base have been deprecated that are already listed. Please see sections on the changes related to the *DtBoundingVolume* class for more information on functions deprecated as a result of the *DtBoundingVolume* refactoring.

DtSpatialIndex

The *DtSpatialIndex* class has been deprecated. It currently remains in the source tree as a test against the new *DtSpatialSubdivision* framework (only for terrain spatial organization though). The *DtTerrainDatabase* no longer contains a *DtSpatialIndex*, and instead now contains a *DtSpatialSubdivision*. The life cycle functions (constructors, and so on) of the *DtSpatialSubdivision* are the same or extremely similar to the *DtSpatialIndex*. As a result, changing code that created a *DtSpatialIndex* to create a *DtSpatialSubdivision* is the following:

Existing method

```
DtSpatialIndex* spatialIndex = new DtSpatialIndex(params...);
```

New method:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision =  
    new DtSpatialSubdivision<DtGdbNode*>(params...);
```



The spatial subdivision is a templated container. As such the largest difference between creating/using the spatial index and the spatial subdivision is the need to specify the template parameter. For more information about templates, please consult a C++ reference book.

Changes to *dyad.h* and *dyad.cxx*

The code has been updated to account for new changes in the use of bounding volumes (and intersecting them). Additionally, the code in the *dyad.h* and *dyad.cxx* files was refactored to improve understandability as well as correctness. Mostly, all members (with very few exceptions) have been renamed from `_varname` to `myVarName`. As many of these are public or exposed to friend functions, this helps tremendously when dealing with the classes as it is clear what is member data and what is not.

Changes to *intersection.h*

The functions in *intersection.h* were removed, except for `GtIntersectionUniqueSolution()`, which was moved into new *mathUtilities.h* and *mathUtilities.cxx* files in the geometry project. The functions were not used anywhere in the code base and were outdated.

Additionally, `GtIntersectionUniqueSolution()` was renamed `DtIntersectionUniqueSolution()` to be consistent with the source code, and the last two parameters were removed from the function as they were not used. To update existing references to this function, simply remove the last two parameters (and rename the function).

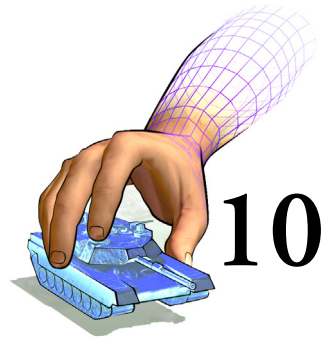
Addition of *geometricUtilities.h*

A new file utility functions for geometric calculations has been added to the geometry project (*geometry.lib*). (This file may eventually be moved.) The file contains functions to calculate an extent from a bounding volume - either by the bounding volume's radius, or by the actual transformed bounding volume (bounding volume in the reference coordinate system). For more information on how this calculation is performed, see the bounding volume header file. These functions are used by objects to create an extent from their bounding volumes, for use in intersecting with a spatial subdivision.

9.8.19. New Examples

VR-Forces has the following new examples:

- ♦ `envProcessListen` demonstrates how to listen for and display state updates for environmental process objects (control and overlay objects) generated by VR-Forces.
- ♦ `addAttr` demonstrates how to extend the RPR FOM. It is derived from the VR-Link `addAttr` example.
- ♦ `chat` demonstrates how to derive a GUI remote control class and install it.
- ♦ `interfaceContent` - example of sending a new type of *DtSimInterfaceContent* subclass which contains a string and number from the front-end to the back-end and vice versa.



Migrating from 3.8 to 3.9

This chapter contains the instructions for moving from VR-Forces 3.8 to VR-Forces 3.9.

10.1. GUI API Changes

This section describes changes to the GUI API.

10.1.1. Showing and Hiding Symbols

Because of the ever increasing number of filters and ways to show or hide symbols, an optional parameter has been added to the `show()` and `hide()` member functions that dictate which module or feature is trying to show or hide a symbol. The `setKeepHidden()` member function was an effort to try to keep symbols hidden, however, it was not successful, because there were too many filters or features that wanted to keep a symbol hidden. So, for release 3.9, `setKeepHidden()` has been removed.

The *symbolModules.h* file contains the list of current features and modules that may want to hide symbols:

```
define DtNewSymbolModule(newType, value) const DtSymbolModule newType =  
    (DtSymbolModule)((int)DtUserSymbolModuleStart + value);  
enum DtSymbolModule {  
    DtAnySymbolModule = -1,  
    DtViewDriverSymbolModule = 1,  
    DtAggregateFilterSymbolModule = 2,  
    DtDragDropHandlerSymbolModule = 3,  
    DtVertexEditorHandlerSymbolModule = 4,  
    DtCreationHandlerSymbolModule = 5,  
    DtForceSymbolModule = 6,  
    DtCircleResizingHandlerSymbolModule = 7,  
    DtOverlayFilterSymbolModule = 8,  
    DtEmbarkationSymbolModule = 9,  
    DtHealthFilterSymbolModule = 10,  
    DtEntityTypeFilterSymbolModule = 11,  
    DtSymbolUpdaterSymbolModule = 12,  
    DtViewObjectsFilterSymbolModule = 13,  
    DtUserSymbolModuleStart = 1000  
};
```

The default module *DtAnySymbolModule* acts as show/hide does today. If you call `hide()` with that module, it tries to hide the specified symbol. However, the symbol might be shown by another filter or feature. If you call `show()` with that module, the cache of requests to hide the symbol is cleared and the symbol is displayed. However, the symbol can subsequently be hidden by another call to `hide()`.

The *fogOfWar* example has been updated to show how to use this new feature. The following code snippet is called when a new symbol is created. It either adds the symbol to the display or hides it, depending on the value of the variable “show”.

```
DtNewSymbolModule(DtFogOfWarSymbolModule, 0);  
if (show)  
{  
    if (toEffect->hiddenByModule(DtFogOfWarSymbolModule))  
    {  
        toEffect->show(DtFogOfWarSymbolModule);  
        // Only do this if we are using the main map area  
        if (myMapArea && dlg)  
        {
```

```

        fApp->removeFogOfWarItem(modelData);
        changed = true;
    }
}
else
{
    if (!toEffect->hiddenByModule(DtFogOfWarSymbolModule))
    {
        toEffect->hide(DtFogOfWarSymbolModule);
        // Only do this if we are using the main map area
        if (myMapArea && dlg)
        {
            fApp->addFogOfWarItem(modelData);
            changed = true;
        }
    }
}
}

```

Check your code closely for any calls to `show()` or `hide()` and take advantage of this new parameter. If you want your symbol to stay hidden, define a new module for hiding it by and call `hide()` with that module. The symbol will not be shown again until all modules that have hidden the symbol have called a matching `show()` on it, so, if you always want the symbol to stay hidden, call `hide()` on it once with your module and do not call a matching `show`.

Also, if you have code that looks like this:

```
if (mySymbol->isShown()) { mySymbol->hide(); }
```

remove the `->isShown` check, because you want the symbol to know that you request it to stay hidden. Until you register your module with the symbol, it could be shown at any time.

If you want to know whether or not your module has been added to the symbol to hide it, you can call the symbols `hiddenByModule()` member function with your module identifier. If it returns true, it has hidden it (see the `fogOfWar` example).

Notes

- The `show()` and `hide()` functions now use a module "*DtFogOfWarSymbolModule*" to tell that symbol that the fog of war filter wants it to stay hidden, and a matching call to `show` when the fog of war filter wants to remove it from consideration from hiding the symbol.
- Multiple calls to `hide` a given module have no effect (the module will only be hidden once.) Similarly, a call to `show` a module that is visible has no effect.

10.1.2. Overlay View Manager

The `setOn()` member function and `layerOnStateChanged` signal are no longer used by the Overlay Manager (and all related overlay items). They have been replaced by `setEditable()` and `layerEditableStateChanged()`.

10.1.3. New View Menu Action Groups

New action groups have been added to organize the view menu:

- myTerrainControlDisplayGroup organizes all terrain related view items
- mySimulationControlDisplayGroup organizes all simulation related items.

10.1.4. Help Menu

The enum for the Help Menu has been changed from DtHelpMenu to DtAppHelpMenu.

10.1.5. Changes Made to Implement the Objects Toolbar

The Objects toolbar consolidates several formerly independent entity and object views and adds new ones.

The Echelon Panel is now Part of the Objects Toolbar

The Echelon View has been integrated into the Objects toolbar. It also now supports display of embarkation relationships. If you have written code to make the Echelon Panel dockable, you can remove it. The following classes have changed:

- The *DtPvdEchelonPanel* class has been removed and replaced with *DtPvdEchelonView*
- The *DtVrfEchelonPanel* class has been removed and replaced with *DtVrfEchelonView*
- In the *DtVrfGuiEchelonViewHandler* the **myEchelonPanel** member variable has been replaced by **myHierarchyPanel** to prevent confusion with class name.

If you have overridden code to create your own version of the echelon tree, panel, or view, you must update your code. You do not need to change any code as part of your EchelonTree subclass, because the changes are mostly reorganization to account for the new Embarkation View.

Previously, you needed to subclass the echelon panel to create a new echelon tree. Now, you subclass the echelon view. So, if your code looked like:

```
# ifndef MYECHELONPANEL_H_
# define MYECHELONPANEL_H_
# include "vrfEchPanel.h"

class MyEchelonPanel : public DtVrfEchelonPanel
{
    Q_OBJECT
public:
    MyEchelonPanel( QWidget* parent = 0, const char* name = 0,
        WFlags fl = 0 );
    static DtTMEchelonPanel* create( QWidget* parent = 0,
        const char* name = 0, WFlags fl = 0 );
protected:
    virtual DtTMEchelonTree* createEchelonTree(QWidget* parent);
```

```

};

# endif
# include "myEchelonPanel.h"
# include "myEchelonTree.h"

MyEchelonPanel::MyEchelonPanel( QWidget* parent, const char* name,
    WFlags fl) :
    DtVrfEchelonPanel(parent, name, fl)
{ }
DtTMEchelonTree* MyEchelonPanel::createEchelonTree(QWidget *parent)
{
    return new MyEchelonTree(parent);
}
DtTMEchelonPanel* MyEchelonPanel::create( QWidget* parent,
    const char* name, WFlags fl)
{
    return new MyEchelonPanel(parent, name, fl);
}

```

it should now look like:

```

# ifndef MYECHELONVIEW_H_
# define MYECHELONVIEW_H_
# include "vrfEchView.h"

class MyEchelonView : public DtVrfEchelonView
{
    Q_OBJECT
public:
    MyEchelonView(const DtEchelonHierarchy *hier,
        const DtEchelonViewOptions *opt = NULL);
    static DtEchelonView* create(const DtEchelonHierarchy *hier,
        const DtEchelonViewOptions *opt = NULL);
protected:
    virtual DtTMEchelonTree* createEchelonTree(QWidget* parent);
};

# endif
# include "myEchelonView.h"
# include "myEchelonTree.h"

MyEchelonView::MyEchelonView(const DtEchelonHierarchy *hier,
    const DtEchelonViewOptions *opt) :
    DtVrfEchelonView(hier, opt)
{ }
DtTMEchelonTree* MyEchelonView::createEchelonTree(QWidget *parent)
{
    return new MyEchelonTree(parent);
}
DtEchelonView* MyEchelonView::create(const DtEchelonHierarchy *hier,
    const DtEchelonViewOptions *opt)
{
    return new MyEchelonView(hier, opt);
}

```

Also, instead of creating the echelon panel through the factory, you now do this:

```
factory.setEchelonViewCreatorFcn(MyEchelonView::create);
```

The Entity List and Environmental List are Toolbars

The entity and environmental lists are now part of the Object toolbar. If you have written code to make these items dockable, you can remove that code. As part of this change, the entity list and environmental list handlers have been removed and replaced with a factory created *DtObjectsList* class. This item is created as part of window creation and initialization.

10.1.6. Tdb and Papermap Drawers

The *DtTdbDrawer*, *DtQtDrawer*, and *DtPaperMapDrawer* classes have been moved from the *DtTerrainApplication* level to be created as instances of the terrain background. These objects used to exist at the application level (that is, one instance per application.) They are now created at the map level (one instance per map.) Creation of these objects has been moved to the factory. If you have subclassed these objects and created them as a subclass of the application, you can install a new static `create()` member function, as follows:

```
factory.setTdbDrawerCreatorFcn(MyTdbDrawer::create);
```

No other code in your subclass needs to be changed.

10.1.7. *DtTerrainLocationEntryWidget*

The *DtTerrainLocationEntryWidget* class's `init()` member function now takes the coordinate system collection as the first parameter. This can be referenced as `DtTMApplication::app()->coordinateSystemCollection()`.

10.1.8. Object Mapping Classes

The lookup functions in *DtMappingList* and *DtAddressMap* have been changed to return a boolean for success of the lookup and to put the result in an address parameter. This was done to allow a null simulation address to be put into the map to support remapping of missing object maps.

10.1.9. Setting Font Size on Linux

The font size was hard coded in `DtPvdApplication::initApplicationFontSize()`, which was removed in favor of loading the font specifications using `QSettings` in `DtTMApplication::initFromQSettings()`.

10.1.10. Overlay Objects can be Considered Control Objects

The *DtPvdApplication* now has functions to specify which entity mapper keys are considered to be control objects. This can be done by calling the `addControlObject()`, `removeControlObject()` and `initControlObjects()` member functions. This information will be use when determining which control objects can be parented on overlay layers locally or are initially published on the "map area".

10.1.11. *DtNameLabelSymbol* Adds Layer Member

Since all symbols can now be placed on overlays, the *DtNameLabelSymbol* now has a "layer" symbol associated with it.

10.1.12. `dropEvent()` Renamed

The `dropEvent()` member function in *DtVrfGuiDragDropHandler* has been renamed to `dropSymbols()`. All code can stay the same.

10.1.13. Terrain Selection Changed

Terrain selection has changed from organizing selected vector features one-to-one (terrain edge to selected terrain edge and terrain node model data to selected terrain node) to organizing around fid codes of the features. Each fid code selected will get a *DtTerrainModelData* representation and, that model data will contain all the features selected that match that fid code. When drawn, composite symbols are created to contain a single symbol for each terrain feature selected by fid code.



The *DtTerrainEdgeModelData* and *DtTerrainNodeModelData* classes have been removed.

10.1.14. Changes to *DtTerrainLineConfigWidget*

DtTerrainLineConfigWidget used to maintain copies of all line symbols that were being edited in order to correctly highlight segments. Because of changes in how terrain segments are kept in model datas, this has been simplified to a single instance of a line symbol that will configure itself to cover the entire selected edge. If you want to override this behavior, you can override the new `setupEdgeSymbol()`, `setupEdgeModelData()` and `selectEdge()` member functions.

10.2. Terrain API

This section describes changes to the Terrain API.

10.2.1. Openflight Reader

The OpenFlight reader now loads:

- Mesh nodes with triangle strip primitives. (Triangle strips are the only primitive available and supported by Multigen-Paradigm at this time.)
- Maps with both localized and unlocalized origins.

An OpenFlight terrain database may be composed of multiple OpenFlight files (or tiles.) In unlocalized databases, all OpenFlight files are built with respect to one origin. The OpenFlight reader loads terrains with “Local Origin Offset” set to zero or set to the same value for all the tile files.

In localized databases, each OpenFlight file contains its own origin information. The “Local Origin Offset” values can be different in the tile files. In the [Figure 10-1](#), each cell represents a terrain file. The dot represents an origin.

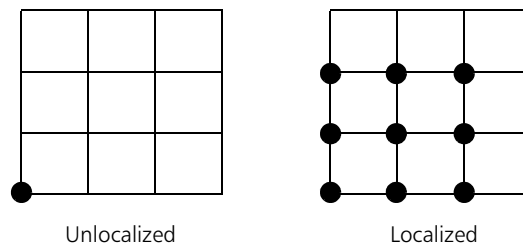


Figure 10-1. Unlocalized and localized terrains

10.2.2. MetaFlight Reader

VR-Forces can now load MetaFlight files. The MetaFlight reader has the following abilities and restrictions:

- ♦ It loads maps with both localized and unlocalized origins.
- ♦ Coordinate system information is extracted from the first Openflight file loaded.
- ♦ It only loads files that contain a GeometryGridDataset node. FileListDatasets, VirtualTextureDataset, ImageGridDataset, VectorGridDataset, and GeneralGridDataset are not supported.
- ♦ It loads the highest level of detail available.
- ♦ It does not support paging.
- ♦ MetaFlight files use a token replacement scheme to define the path to files. The tokens identify a data item that can identify an element inside the file. The loader processes the DSNAME, DSTITLE, LEVEL, COL, and ROW tokens. It also checks for the MFT_DIR environment variable, which is a variable set by Vega Prime.
- ♦ If the <Volume> and <Rootpath> entries in the <GeometryGridDataset> tag are set, the loader uses their values to build a full path to search for the files. If they are empty, the loader looks for the files relative to the location of the .mft file.

10.2.3. Changes to Shapefile Support

A new class, *DtShapeProjection*, contains a tree list of *DtShapeProjectionInformation*. This information is read from the shape .prj file and is used to convert a shapefile created in one projection to the current database projection. If a projection type is not supported by the *DtShapeProjection* class, you can subclass this class and install the creator to be created in the *DtShapefile* class. See the *DtShapeProjection* header file (*shapeProjectionInformation.h*) for the member function to override if you want to perform your own coordinate projection.

10.2.4. Changes to GDB

Coordinate system nodes now save out spheroid information (including datum shift). The UTM_Coordinate_System also saves out the Transverse Mercator scale factor.

10.2.5. FltMetafileRecord and DfdDfadMetafileRecord Additions

The following items have been added to the FltMetafileRecord and DfdDfadMetafileRecord record types to allow for definition of a reference ellipsoid:

- ♦ Reference_Ellipsoid_Semi_Major_Axis
- ♦ Reference_Ellipsoid_Semi_Minor_Axis
- ♦ Inverse_Flattening
- ♦ DatumShift x y z
- ♦ Projection_Central_Scale.

If both Inverse_Flattening and Reference_Ellipsoid_Semi_Minor_Axis are defined, the Inverse_Flattening value takes precedence and the minor axis is calculated as:

$$\text{semiMinor} = \text{msemiMajor} - (\text{semiMajor} / \text{Inverse_Flattening})$$

10.2.6. DtExtentInitializer Respects Z Values

The *DtExtentInitializer* class now respects the altitude (Z) values of the specified extent in the *DtExtentMetaFileRecord*. The altitude of the vertices is determined by the altitude of the left, back, bottom vertex and the right, front, upper vertex. These are used as the SW and NE vertices respectively. The altitude of the SE vertex matches the altitude of the SW vertex and the altitude of the NW vertex equals the altitude of the NE vertex.

Figure 10-2 illustrates the polygons that get created from the extents.

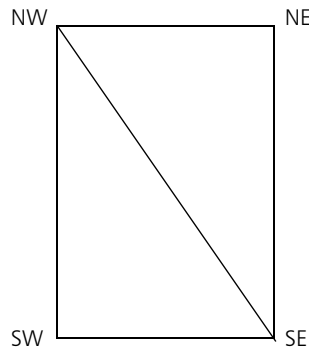


Figure 10-2. Extent polygons

You can now specify a maximum latitude or longitude that is smaller than the minimum latitude or longitude. VR-Forces wraps around until it is larger. The extent initializes the maximum to be within +/- 180 or 90 respectively (for example, min = 150, max = -150).

For example, this allows simulation of the area covered by +150 degrees to -150 degrees.

10.2.7. GDB File Format Changes

The GDB now writes out, and reads, in spheroid and shift information. The UTM_Coordinate_System now writes out scale factor information.

10.2.8. *DtGdbNode* Class Hierarchy Interface Changes

The allInstancesOf() member function has been changed to take a std::set as the container instead of a std::list. This ensures that no element being accumulated can be counted more than once.

10.2.9. *DtTerrainDatabase* Interface Changes

In the API const member functions that returned non-const references or pointers to internal members have been replaced with two functions, a const version and a non-const version, which return a const and non-const pointer or reference respectively.

This is an important change for the following reasons:

- Const member functions should never return a non-const reference or pointer to an internal member, as that reference or pointer can then be used to modify the const object, which is an error and incorrect.
- To maintain consistency in the interfaces to our objects, it was important to standardize the accessors.
- In preparation for other concerns, such as multithreading, it was important to better define when objects can be modified and when they cannot be modified.

The most likely result of these changes is that the `coordinateSystem()` function will be the source of new compile errors. In many places in the VR-Forces code base, a non-const coordinate system pointer was retrieved from a const terrain database. To fix compile errors, define the temporary object, member variable, or function parameter as a const coordinate system pointer instead of a non-const pointer.

10.2.10. Coordinate System Class Hierarchy Changes

The coordinate system classes have been renamed, with the exception of the base *Coordinate_System* class. This has been done to clean up inconsistencies in the names compared to the rest of our code base and to help with their clarity and usability.

10.2.11. Library Changes

The *libproj* library has been added to do projection conversions for geotiff paper maps.

10.3. Simulation API Changes

This section describes changes to the Simulation API and Remote Control API.

10.3.1. The Physical World

VR-Forces 3.10 introduces the concept of the physical world. It encompasses anything with a physical presence in the virtual world. Practically speaking, this means any object that has a bounding volume – entities, terrain polygons, vector obstructions, and environmental objects. The physical world model is required for embarkation. It also results in improved efficiency for line-of-sight calculations and obstruction avoidance. All models have been updated to work with the physical world. VR-Forces is backwards compatible with the older models, but you must update to the physical world to take advantage of its improvements.

The physical world is implemented through the `DtPhysicalWorld` class. You can extend the API to change what is considered to be part of the physical world.

10.3.2. Object Parameters Database and Related Configuration Changes

This section lists changes to the object parameters database and how entities are configured.

- ♦ The hit probabilities for the M16 and AK47 rifles have been updated to be much lower to better reflect the probability of hitting the target when firing either of those weapons. This is a behavior change, and may result in slightly different scenario behavior.
- ♦ The pseudo-aggregate-move-actuator descriptor has been removed from all aggregate *.ope* files because the component corresponding to this parameter entry has been removed. Its functionality has been replaced by the pseudo-aggregate-extent-actuator.
- ♦ Entity OPE files configured with a target-detect-sensor have a new parameter named `target-all-priority-lists`. The meaning of this parameter is described in [“New Parameter target-all-priority-lists Added to Target Detection Sensor,”](#) on page 10-15.
- ♦ The F-117 configuration files are now separated from the generic attack plane, and include reduced radar signature compared to other planes.

Firepower and Mobility Kills

All basic VR-Forces models now check the firepower and mobility kill appearance bits, and will not fire weapons or move if the respective bits are true.



Mobility kills are treated by VR-Forces models as failures of the drive system of the entity, so they will coast to a stop, or whatever is appropriate for the entity type, instead of stopping immediately.

New Sensor Model System

A new method for modeling sensors within VR-Forces has been added, called the Signature Sensor System. Now, entities are configured with signature values for various sensor domains, such as visual, radar and infrared. Propagation models are configured which model how these signatures decrease over distance, and sensors configured on other entities use these signature to determine what entities they can see.

VR-Forces is configured with Visual, Radar, Infrared, and Sonar sensor domains by default. You can add your own sensor domains through the configuration files. For details, see “[Sensors](#),” on page 5-37 in *VR-Forces Developer’s Guide* and “[Configuring Sensors](#),” on page 8-2 in *VR-Forces Configuration Guide*.

Improved Multisensor/Multiweapon Capability

The *DtTargetDetectionSensor* has been replaced with *DtSignatureObjectSensor* (and related sensors) and a new component called *DtTargetSelectionController*. It is possible to hook as many sensors as desired to the target selection controller, and to have the target selection controller choose target priorities independently for as many different weapon systems as desired. It can choose targets for each weapon based on target type, relative position of the target to the entity, or a combination of both. Using this, it is much easier to configure multiple weapon systems on a single entity.

Spot Report Generator

A new component, *DtSpotReportGeneratorController*, is included with VR-Forces. It generates and sends spot report radio messages based on the outputs of sensors configured on an entity.

New and Changed Entity Components

New Parameter `target-all-priority-lists` Added to Target Detection Sensor

The weapon and launch controllers no longer target entities for which they do not have ammunition. This change allows you to target entities in lower priority lists that an entity might have ammunition for.

The Target Detection Sensor has a new parameter, `target-all-priority-lists`. When `target-all-priority-lists` is false (the default) if a target is found in a priority list, then lower priority lists are not checked, regardless of whether or not the entity has the resources to fire at the identified target. Therefore, if an entity runs out of ammunition for higher priority targets, but still has ammunition suitable for lower priority targets, it might never use that ammunition because high priority targets are still being found and the lower priority lists are not being checked.

If this value is true (default is false), then all entity lists are checked for possible targets. Also, an entity will fire at low priority targets that are in its sector of responsibility before it fires at high priority targets that are not in its sector of responsibility. However this setting degrades performance. Use this option only if you know that you have an entity that has multiple weapons for a single controller that might want to target lower priority entities when it runs out of ammunition for higher priority entities.

New Parameters for the Turret Scan Controller

The Turret Scan Controller has two new parameters, `minimum-scan-width` and `scan-sor`. `minimum-scan-width` specifies the minimum width (in radians) of the sector of responsibility (SOR) within which the turret will scan back and fourth. If the SOR width is smaller than this value (default to 10 degrees), with turret will be commanded to a stationary position in the center of the SOR. When the SOR is larger, the turret will scan within the sector, as before.

`scan-sor` specifies whether or not the Turret Scan Controller should actively slew the turret over the current sector of responsibility (old behavior) or should recenter the turret and hold when the turret is not otherwise in use. Default: true.

The Rotary-Wing and Subsurface `max-altitude` Parameter has Moved

In rotary-wing and subsurface entities, the `max-altitude` parameter has moved from the flight-kinematics descriptor to the state repository parameters. This allows the actuator and state repository to clamp the object to the `max-altitude`. This change affects the following classes:

- ♦ *DtRotaryWingEntityParameters* – adds `maxAltitude()` and `setMaxAltitude()`
- ♦ *DtRotaryWingActuatorDescriptor* – removes `maxAltitude()`, `setMaxAltitude()`, and the `myMaxAltitude()` variable, which is now referenced from the state repository.

flaps-deceleration Parameter Added to Fixed-Wing Model

The fixed-wing model has a new parameter, **flaps-deceleration**, which specifies the additional deceleration caused by deploying the flaps on the fixed-wing entity. The fixed-wing pilot controller will now deploy the flaps when it is attempting to slow down significantly.

Entities Resume Tasked by Superior Status Upon Task Completion

By default, entities now resume being tasked by superior (equivalent to clearing their independently tasked status) when they complete an independent task. In previous releases, entities would retain their independently tasked status indefinitely after completing a task. The only way to set an entity to be tasked by its superior again was to send a *DtSetTaskedBySuperior* set data request message to the entity (or select Tasked by Superior from the VR-Forces GUI). Now it will resume tasked by superior status immediately upon task completion.

You can override this new default behavior and revert to the old behavior (entities retain their independently tasked status upon task completion) by setting the new **tasked-by-superior-upon-task-complete** parameter to false.

Parameters for Fixed-Wing Landing

A fixed-wing landing approach route is created from the following descriptor information:

- ♦ **approach-distance**: The distance from the runway at which the approach route should end.
- ♦ **approach-altitude**: The height above the terrain at which the approach route should end.
- ♦ **landing-speed-ratio**: The percentage of lift the plane's speed should be in order to land. Default: 5% greater than lift.
- ♦ **publish-approach-route**: Specifies whether the approach route should be published on the network. Default: true.
- ♦ **landing-descent-rate**: The rate at which a plane can descend, in meters per second. This parameter is used to calculate the length of approach from the entity's altitude.
- ♦ **landing-gear-down-distance**: The percentage away from the end point at which to turn on landing lights (which puts down the landing gear).

Parameters for DtFixedWingTakeoffController

The point to move to when a fixed-wing entity takes off is determined by the following new parameters:

- ♦ **takeoff-target-distance**: The distance away from the takeoff route to fly to after takeoff
- ♦ **target-altitude**: The height above the terrain to fly to.

10.3.3. Embarkation Controller

New controllers to handle embarkation have been added to the back-end (along with associated tasks and sets to handle embarkation and disembarkation). There is now a base level embarkation controller (*DtEmbarkationController*) (*embarkationController.h*) that handles both the embarkation of an object onto a parent and the disembarkation from that parent. The *DtEmbarkationController* is a state machine that handles the embarkation of an object from its current location to an embarked location. See the class documentation for *DtEmbarkationController* for details about how it works.

Controllers Derived from DtEmbarkationController

The following controllers have been subclassed from *DtEmbarkationController* to add the following specific overriding behavior:

DtRotaryWingEmbarkationController

DtRotaryWingEmbarkationController overrides the *DtAtEmbarkationPoint()* member function to immediately set the state to *DtStoppedAtEmbarkationPoint*, because rotary-wing entities do not recognize the *DtStopMoving* task.

DtAggregateEmbarkationController

DtAggregateEmbarkationController passes tasks to currently taskable subordinates. Sets are passed down to all subordinates. The controller also adds the *checkEmbarkationState()* member function, which is called every tick regardless of whether or not the aggregate is tasked. The pseudo-aggregate is considered embarked if all available taskable subordinates are embarked. The aggregate is embarked on the same object as the current leader of the aggregate. If any subordinate disembarks, the pseudo-aggregate also disembarks.

DtOccupancyDirectorController

DtOccupancyDirectorController allocates reserved embarkation slots and determines which (if any) objects can be embarked on an object. This controller responds to messages of type *DtIfRequestEmbarkationInformation*. When the request is received, the controller looks to see if there are any slots available, and, if so, reserves it (if requested) and then returns a *DtIfEmbarkationInformation* structure with the reservation information. This controller also provides disembarkation information and responds to disembarkation requests. If an entity cannot embark either because the object does not support the requesting object type or because the slots are full, an information return structure is sent with the *DtEmbarkCannotEmbark* result status.

DtAggregateOccupancyDirectorController

DtAggregateOccupancyDirectorController passes the request for embarkation to the leader of the aggregate. The leader of the aggregate is then responsible for determining who can embark upon it and where.

DtFixedWingLandingController

DtFixedWingLandingController creates the approach route and controls the landing of the plane. See [“Parameters for Fixed-Wing Landing,”](#) on page 10-16 for a list of parameters that specify the approach route.

DtFixedWingTakeoffController

Like the landing controller, this controller tasks a plane to move along the takeoff route using the *DtFlyAlongRoute* task. During the taxiing, the plane’s speed is set to be the takeoff speed (default: 30% above lift). When it reaches the end of the route, the plane is tasked to fly to a point above the ground, using *DtFlyToLocationTask*. (*DtFlyToLocationTask*, like *DtFlyAlongRouteTask* also has the `useFixedPitch` flag, which is also true).



No information needs to be published since the plane is simply flying to a point. Once the task is complete, if the plane has no other task it circles around the takeoff target point.

Changes in Models and State Repositories Due to Embarkation

Because entities can now be embarked on other entities, and can be tasked to perform operations on those entities, if you model dynamics of an entity, you must change your models to take advantage of this behavior.

i

Any existing code that assumes that entities are only able to be attached to terrain will continue to work, but this code will not work with the embarkation feature. We recommend that you update your code to remove this assumption.

You cannot assume that an entity is solely attached to the terrain. An entity can now be attached to any other model (except for environmental objects) in the system. When an entity is embarked, you need to know the following information about an entity:

- ♦ Terrain intersections
- ♦ Velocity relative to the host
- ♦ Acceleration relative to the host.

The following code changes need to be made in order to account for those three items:

- ♦ Terrain Intersections:
In your models, if you reference `mySimTerrain->terrainDatabase()`, change the reference to `myEntity->terrainDatabase()`. The *DtVrfObject* now has a pointer to a *DtObjectGeometricModel* object. This object wraps both the base terrain (`mySimTerrain->terrainDatabase()`) and any attached entity. If the object in question is attached to another entity, then an intersection will first be done on the attached entity. If an intersection is found, that will be returned, else the sim terrain will be checked. If the entity is not attached to another entity, then only the sim terrain will be checked. Note that even though this model is subclassed from *DtTerrain*, it is done that way only for the API. Any terrain calls made to that object directly to try and deform the terrain will ultimately fail.
- ♦ Velocity relative to the host:
New member function calls, `relativeVelocity()` and `setRelativeVelocity()` replace any references you might have to `localVelocity()` and `setLocalVelocity()`. The return and set values are still in local (database) coordinates, but if the entity is attached, the return value of `relativeVelocity()` will be the velocity of the entity only, whereas if you call `localVelocity()`, it will be the sum of the velocity of the entity and any objects it might be attached to. You must also call `setRelativeVelocity()` to preserve the notion that this velocity is for the entity only. If the entity is not attached, these calls will pass through to `localKinematicState().velocity()` and `localKinematicState().setVelocity()`;
- ♦ Acceleration relative to the host:
Same as velocity relative to the host, except calls are named `relativeAcceleration()` and `setRelativeAcceleration()`.

10.3.4. Attaching Environmental Objects to Entities

You can now attach environmental objects (routes, waypoints, areas, and so on) to entities. For example, a route can be placed on an aircraft carrier for a DI to traverse. To support this, an optional parameter (initially the empty string) has been added to `createOverlayObject()` (before the `userData` pointer) that can contain the name of the object to attach to. The `sendVrfOverlayObjectModifyMsg()` member function also has an `attachTo` and `keepExistingAttachment` parameters. This is so the create can assign the attachment and future modifies can keep any attachment that has been created (or, can be changed).

10.3.5. Initializing Extended State Repositories

The functionality of extending state repositories at the base class level has been expanded to allow initialization of the state repository extensions from an object parameters database file.

Previously, you could create derived *DtVrfStateRepositoryUserExtension* classes, which extended the base *DtVrfObjectStateRepository*. These derived classes were a logical extension to an entity's state repository, and therefore were saved and restored as part of the entity in the order of battle file. This capability has been extended to enable you to initialize your derived *DtVrfStateRepositoryUserExtension* from an OPE file.

To take advantage of this capability, you can configure initial values for the readable/writeable data members of your derived *DtVrfStateRepositoryUserExtension* by adding a user-extension parameter entry to an entity's parameter database file. For example, if you have created a derived *DtVrfStateRepositoryUserExtension* of type "my-state-repository-user-extension", that has a single readable/writeable string data member with the *DtReaderWriter* name "my-tag", you could configure its initial value in the OPE file as follows:

```
(state-repository-extension-type "my-state-repository-user-extension")
(user-extension
  (my-tag "some-initial-value")
)
```

To implement this new functionality, some changes to the existing *DtVrfStateRepositoryUserExtension* class API were needed. If you have implemented a derived *DtVrfStateRepositoryUserExtension* classes, you must make the following modifications to your derived class:

- Constructors and the `create()` member function now take a pointer to a *DtVrfObjectStateRepository** as an argument, instead of a reference.
- You must now also implement a `clone()` member function in your derived class.

10.3.6. Assume Tasks Have Been Removed

All of the assume tasks in VR-Forces have been removed, and all code that requests, sends, and processes these tasks has been removed. The assume tasks were used in the aggregate models in versions of VR-Forces prior to version 3.8. In version 3.8, the aggregate models were completely redesigned. The assume tasks and the mechanism for reporting and handling these them are no longer needed or used.

The following task classes have been removed:

- ♦ *DtAssumeMoveTo* (*asmMoveTo.h*)
- ♦ *DtAssumeMoveAlong* (*asmMoveAlong.h*)
- ♦ *DtAssumeFollowEntity* (*asmFollowEnt.h*)
- ♦ *DtAssumePatrolRoute* (*asmPtrlRoute.h*)
- ♦ *DtAssumePatrolTwoPoints* (*asmPatrolPts.h*)
- ♦ *DtAssumeWait* (*asmWaitTask.h*)
- ♦ *DtAssumeWaitDuration* (*asmWtDurTask.h*)
- ♦ *DtAssumeWaitElapsed* (*asmWtElpTask.h*)
- ♦ *DtAssumeMoveToLocationTask* (*asmMvToLocTask.h*).

If you have overridden the member function `DtControllerComponent::reportTaskForAssumption()`, you should remove it from your code.

10.3.7. Changes Supporting Plan Language Improvements

In order to support self-referencing plans and pattern matching conditional statements, the following changes were made

- ♦ The `myModifier`, `myAnyFlag` and `myResult` members and their associated accessors and mutators were moved down to *DtSimSubordinateSearch* from all its subclasses. Any user-created *DtSimSubordinateSearch* subclasses should also remove these members, using the `conditionSim` example as a guide. The following subclasses were affected:
 - `DtEntDestroyedSearch`
 - `DtEntInAreaSearch`
 - `DtEntLeftOfLineSearch`
 - `DtEntHasTargetSearch`
 - `DtEntUnderFireSearch`
 - `DtEntEmbarkationStateChangedSearch`
- ♦ *DtEvaluator* subclasses call the new `DtSimSubordinateSearch::search()` member function from their `evaluate()` member functions. This new function supports more complex modifiers in order to support self-referential and pattern searches. Any user-created *DtEvaluator* subclasses should update their `evaluate()` member functions, using `MyEvaluator::evaluate()` from the `conditionSim` example as a guide. The following subclasses were affected:
 - `DtEvalEntDestroyed`
 - `DtEvalEntEmbarkationStateChanged`
 - `DtEvalEntHasTarget`
 - `DtEvalEntInArea`
 - `DtEvalEntLeftOfLine`
 - `DtEvalEntUnderFire`.

10.3.8. New and Changed Component Classes

This section describes controllers that have been added or changed. Also, see “[Embarkation Controller](#),” on page 10-17 and “[New and Changed Entity Components](#),” on page 10-15.

***DtAggregateFormationController* Changes**

The `generateListOfSubordinatesEligibleForFormation()` member function now returns a `DtSubordinateQuery` result, which you must free when you are finished with it. `isEligibleForFormation()` has been removed in favor of new subordinate query API. `getCurrentFormation()` now takes a `DtSubordinateQuery` as a parameter, not a vector of strings.

Component Port Architecture Changes

Some *DtInputPort* functionality has been split into *DtSingleConnectionInputPort* to provide flexibility for the port architecture.

If you have created any new input port types by deriving from *DtInputPort*, change your classes to derive from *DtSingleConnectionInputPort* instead. No other changes are necessary.

Joystick Improvements

Many vehicles that did not support joystick control in previous releases are now configured for joystick support. The joystick controllers have been reconfigured with new descriptor classes that allow the axes and buttons used for various controllers to be remapped in the object parameters database. The defaults are still based on the Microsoft Sidewinder Precision 2 joystick, but by changing the mapping parameters you can now configure any type of controller. [Table 10-1](#) lists the new classes and parameters:

Table 10-1: Joystick classes, descriptors, and parameters

Class	Descriptor	Descriptor Name	Parameters
<i>DtGroundJoyMove-Controller</i>	<i>DtGroundJoy-DeviceController-Descriptor</i>	ground-joy-controller-descriptor	♦ joy-steering-axis ♦ joy-invert-steering-axis ♦ joy-throttle-axis ♦ joy-invert-throttle-axis ♦ joy-brake-button ♦ joy-forward-gear-button ♦ joy-reverse-gear-button
<i>DtGroundJoyTurret-Controller</i>	<i>DtTurretJoyDevice-ControllerDescriptor</i>	turret-joy-controller-descriptor	♦ joy-slew-axis ♦ joy-invert-slew-axis
<i>DtBallisticGunJoy-Controller</i>	<i>DtBallisticGunJoy-ControllerDescriptor</i>	ballistic-gun-joy-ctrl-descriptor	♦ joy-fire-button ♦ joy-change-ammo-button ♦ joy-elevation-axis ♦ joy-invert-elevation-axis

Table 10-1: Joystick classes, descriptors, and parameters

Class	Descriptor	Descriptor Name	Parameters
<i>DtFixedWingJoyFlightController</i>	<i>DtFixedWingJoyDeviceControllerDescriptor</i>	fixed-wing-joy-controller-descriptor	♦ joy-pitch-axis ♦ joy-invert-pitch-axis ♦ joy-roll-axis ♦ joy-invert-roll-axis ♦ joy-throttle-axis ♦ joy-invert-throttle-axis ♦ joy-pedal-axis ♦ joy-invert-pedal-axis
<i>DtRotaryWingJoyFlightController</i>	<i>DtRotaryWingJoyDeviceControllerDescriptor</i>	rotary-wing-joy-controller-descriptor	♦ joy-pitch-axis ♦ joy-invert-pitch-axis ♦ joy-roll-axis ♦ joy-invert-roll-axis ♦ joy-yaw-axis ♦ joy-invert-yaw-axis ♦ joy-collective-axis ♦ joy-invert-collective-axis

The *DtBaseJoyDeviceControllerDescriptor* class has been removed. The parameters it attempted to change were global joystick parameters – thus the last controller to set them would set the values for all the controllers. These parameters can still be adjusted in *joyParam.dat*. Any object parameter entries that used base-joy-controller-descriptor must now be updated to use the new descriptor type, which corresponds to the controller type in [Table 10-1](#).

In prior releases, multiple joystick controllers would call *joyDeltas()* on the *DtJoyDevice* to poll the joystick for new events. This was a flawed approach because it allowed one controller to consume events before the next had a chance to examine them. For this reason, *joyDeltas()* is now called only once per frame from *DtJoyDeviceManager::tick()*. The *DtJoyDeviceManager* is ticked from *DtCgf::tick()*.

New Programming Hooks for Handling Unachievable Tasks

The API has new functions that enable developers to write code to handle situations in which a given task is unachievable. For example, say you have a ground vehicle and you give it a task to move to a waypoint, but there is already an entity parked at that waypoint. In this situation, the entity would never be able to successfully reach the waypoint. It would continuously try to reach that point, until it ran out of fuel. Rather than have this entity be forever stuck on this task, you would like to be able to add some reasoning to the entity, so that as it approaches the waypoint, it can evaluate the situation and take some more sensible alternative action, such as parking nearby.

Handling unachievable tasks is done on a per *DtTaskControllerComponent* basis. We have added two new member functions to the *DtTaskControllerComponent* class. These functions provide a place for you to add code for handling unachievable tasks. The functions are `decideToGiveUpTask()` and `giveUpTask()`. The `decideToGiveUpTask()` is a function that returns a boolean flag indicating whether or not the entity should give up its current task. It is called immediately after every tick when the simulation is running (in play mode) and entity is tasked. If it returns true, then it calls `giveUpTask()`. The `giveUpTask()` function provides a place where any special handling, such as calling `taskComplete()`, can be added. By default, the function `decideToGiveUpTask()` returns false, and the `giveUpTask()` function has an empty implementation.

Change to *DtAggregateMoveIntoFormationController*

The `cancelMoveIntoFormation()` member function in *DtAggregateMoveIntoFormationController* has been removed. It was not being called by any code in the VR-Forces back-end.

10.3.9. Handling Formation-changed Events in Aggregate Move-along and Patrol-Along Controllers

The `DtAggregateMoveAlongController` now always processes formation-changed events (generated by the `DtFormationState`), adjusting route-following assignments for its subordinates. Previously, it only performed this processing if it was currently tasked as a top-level task (not a subtask). This change now frees other controllers from having to handle formation-changed events when its using the `DtAggregateMoveAlongController` to carry out a subtask. The `DtAggregatePatrolAlongController` no longer registers for or handles formation changed events (it now relies on the `DtAggregateMoveAlongController` to handle these events). As a result, the following member function have been removed from the `DtAggregatePatrolAlongController`:

```
static void formationChangedCallback(
    const DtFormationState& formationState, void* usr);

static void objectManagerRemoveAndDeleteAllCallback(
    const DtVrfObjectManager& objectManager, void* usr);

virtual void processFormationChanged(
    const DtFormationState& formationState);

virtual void onObjectManagerRemoveAndDeleteAll(
    const DtVrfObjectManager& objectManager);
```

10.3.10. DtVrfObjectPredicate Changes

A new kind of predicate class `DtNullVrfObjectPredicate` (`nullVrfObjPred.h`) has been added. It is a special-purpose predicate that always returns true and is never intended to actually be evaluated. Rather it is used as a sentinel value when manipulating managed lists of objects.

Changed the names of the `DtFilteredObjectList` and `DtFilteredObjectListManager` classes to `DtManagedObjectList` and `DtManagedObjectListManager`, respectively. This was done to better reflect the purpose of these classes. Now that we have a `DtNullVrfObjectPredicate`, it is possible to request a list that is managed (object additions and removals are monitored), but not filtered.

10.3.11. Determining when an Object's Vertices Change

A new callback has been added for determining when a *DtVrfObject's* vertices change relative to one another. This callback is useful when you want to be notified if any of an object's vertices has moved with respect to one another, but the overall object has not moved. You can register for this callback by calling `DtVrfObject::addRelativeVerticesChangedCallback()` (in *vrfObject.h*) in an instance of the *DtVrfObject* you are interested in.

This callback is different from the existing vertex changed callback in *DtVrfObject* (`addVerticesChangedCallback()`). `addVerticesChangedCallback()` invokes callbacks whenever relative vertices change or if the overall object moves.

10.3.12. *DtFormationTable* can be Overridden

You can now override the contents of the formation-table member of the *DtFormationControllerDescriptor* class. This enables you to extend the content of formation tables. The *myFormationMember* is now dynamically allocated. To get the descriptor to read in a derived kind of *DtFormationTable*, you must create a derived *DtFormationControllerDescriptor* and override the *createFormationTable()* member function. In your implementation, have it create and return a new derived kind of *DtFormationTable*. For example,

```
DtFormationTable* DerivedFormationControllerDescriptor::
    createFormationTable(const DtSymbolString& name,
        DtReaderWriterRegistry* parentRegistry) const
{
    return new DerivedFormationTable(name, parentRegistry);
}
```

10.3.13. Support for Simulation Management PDUs in DIS

You can now use DIS simulation management (siman) start/stop PDUs to pause or start a VR-Forces scenario from an external application.

When a start or stop siman PDU is received, the receiving back-end checks the Receiving Entity ID to verify whether it should affect it or not. [Table 10-2](#) lists acceptable values for Receiving Entity IDs

Table 10-2: Acceptable receiving entity IDs

site	host	ID	Result
ALL	ALL	ALL	Everyone (example: 65535:65535:65535)
ALL	ALL	0	All back-ends (example: 65535:65535:0)
n	m	0	Exact ID (example: 1:3001:0)

By default, VR-Forces only reacts to PDUs sent to {ALL,ALL,*} and {ALL,ALL,0}. This is because if VR-Forces has multiple back ends, processing a PDU sent to a specific back-end could leave VR-Forces in an inconsistent state. Processing exact IDs is appropriate for situations in which VR-Forces has only one back-end.

The *simMgmtPduProcessLevel* parameter in *vrfSim.mtl* tells VR-Forces which type of receiving entity IDs to process. [Table 10-3](#) shows how VR-Forces processes start and stop PDUs based on the siman PDU process level.

Table 10-3: Siman PDU processing level

simMgmtPduProcessLevel	Everyone	All Backends	Exact ID
1			X
2	X	X	
3	X	X	X

For example, if the you want VR-Forces to only process siman PDUs sent to a specific back-end {a,b,0}, set `simMgmtPduProcessLevel` to 1.

When a back-end receives a Stop PDU, it stops immediately regardless of the value of the Real-World Time field in the PDU. The Reason and Frozen Behavior fields are ignored.

10.3.14. New, Changed, and Deprecated Examples

- ♦ The `decideToGiveUpTask` example demonstrates how to add a 'give up' capability to a ground vehicle entity's move to waypoint behavior. In the example, the criteria for giving up is a time limit of 10 seconds. If the entity does not reach the waypoint it is tasked to move toward within 10 seconds, it gives up.
- ♦ The Radar Warn Receiver example now uses a Patriot missile battery in its default configuration, which includes a radar publishing emitter system, in place of the customized SAM system. The radar warning receiver and response component are now configured on the red fixed-wing attack/strike entity instead of the blue one. The scanning radar component is no longer part of this example.
- ♦ The Image Splitter functionality has been absorbed into the new `TdbTool`. The Image Splitter example will continue to be provided to demonstrate manipulating terrain through the GUI, but will not be updated with new features to match the `TdbTool`.
- ♦ A sample MetaFlight terrain file has been included in this release – *./data/terrain/Makland-UTM-MFT/Makland-UTM-1-0.mft*.
- ♦ The `convertParamDb` example has been removed, because the format and content of parameter databases has expanded and changed since pre-3.7 versions to too great of an extent for this tool to be useful any longer. If you have parameter databases older than 3.7 that you want to upgrade to the 3.9 version, we recommend that you use the OPD editor to re-create your changes in the 3.9 version of the database.
- ♦ A new example called `spotReport` has been added to demonstrate how to use the new spot reporting capability in VR-Forces. In the example, an aggregate is configured with a new kind of component that listens for and displays spot reports from its subordinates.
- ♦ The new subtask example demonstrates how to use subtasks.
- ♦ Most examples now have a readme file in PDF format that provides an overview of the example.

10.3.15. Miscellaneous API Changes

The *DtPseudoAggregateMoveActuator* Class has been Removed

The *DtPseudoAggregateMoveActuator* class has been removed and its functionality assumed by the *DtPseudoAggregateActuator* class.

Subtasks

The documentation for subtasks has been completely revised. See [“Subtasks,”](#) on page 8-3 in *VR-Forces Developer’s Guide*.

Modify Message Changes

Since all objects can now be part of an overlay layer, the ability to set the overlay parent has been added to the *DtIfModifyVrfObject* message. For overlay objects, setting this via the overlay modify message or the object modify message has the same effect.

Vrf Object Data Message Changes

The *DtIfVrfObjectData* message (*ifVrfObjData.h*) has been changed. It now includes a version number and an overlay parent name

Handling of *DtIfControl* Messages Consolidated in *DtCgfDispatcher*

The handling of *DtIfControl* messages (run, pause, rewind, run for duration) was previously spread out between the *DtExerciseClock* and *DtCgfDispatcher* classes. All processing of these messages is now handled exclusively in the *DtCgfDispatcher* class.

DtFormationState

The *queryAll()* member function now takes a *DtFormationQueryResults* object as an argument.

DtVrfObjectManager

DtVrfObjectManager::writeOrderOfBattleFile() member function used to have the criteria for which entities were written out embedded inside it. These criteria have been broken out into a separate member function, *objectShouldBeSavedToOrderOfBattle()*.

DtScenario

The *save()* member function now takes a boolean value which determines if an object map filename gets saved in the scenario file. If false, the name is set to null (""). This was added to allow the *DtCgf* to perform a local save without having to create an object map.

Fixed-Wing Model Changes

The `processDeleteVrfObject()` and `processWaypointChangedNow()` member functions now take a *DtVrfObject* as a parameter rather than a *DtSimMessage*.

DtBoolean, DtTrue, and DtFalse Deprecated

To be consistent with changes made in recent releases of VR-Link, `DtBoolean`, `DtTrue`, and `DtFalse` have been replaced with `bool`, `true`, and `false`. No code changes are required. The old terminology is type-def'd to the new. You are encouraged to replace instances of `DtBoolean` in your code with `bool`.

Changes to Aggregate Location and Heading

Only healthy (not destroyed or mobility killed) and non-independently tasked subordinates are taken in consideration when computing aggregate location and heading. Version 3.8 included destroyed and independently tasked subordinates when computing average location and heading. See *pdfAggExtent.h* for details.

Individual Lifeforms Posture

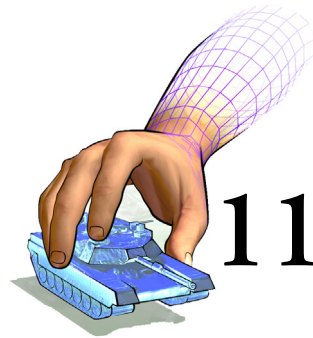
Individual lifeforms now resume their assigned posture as appropriate after dynamically adjusting their posture to handle steep slopes. In the `DtLifeformActuator`, posture is adjusted to `DtLifeformWalking` or `DtLifeformCrawling` each tick, depending on the the current pitch of the terrain and the maximum slope the entity can tolerate. Previously, if the posture had been changed to handle a steep slope, it never resumed the original posture, even if the terrain leveled out again.

Changes to *DtSubordinateManager*

The `objectAboutToBeRemovedCallback()` and `processObjectAboutToBeRemoved()` member functions have been removed from *DtSubordinateManager*. They duplicated work done by *DtOrganizationView*. This change may improve performance.

The `DtCgf::closeScenario()` Member Function Now Pauses the Status Publisher

In previous releases, if the back-end was in run mode, when a scenario was closed, it would pause the simulation clock but leave the scenario in run mode. This could cause inconsistent behavior. Now, when the scenario is closed through the API, the back-end status publisher is put into pause mode.



Migrating from 3.9 to 3.10

This chapter contains the instructions for moving from VR-Forces 3.8 to VR-Forces 3.9.

11.1. API Changes

11.1.1. New Plug-In API Architecture

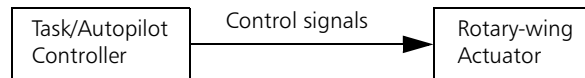
Developers can now create shared libraries that can be loaded at runtime so that you can add functionality to VR-Forces without recompiling the entire application.

For complete details, please see Chapter 1, *Introduction to the VR-Forces APIs*, Chapter 2, *The VR-Forces Simulation Engine*, and Chapter 11, *The VR-Forces GUI API* in *VR-Forces Developers Guide*.

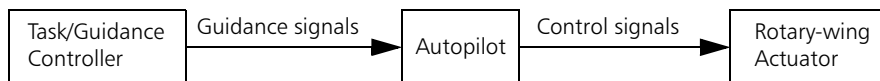
11.1.2. Changes to the Rotary Wing Model

The rotary-wing model has been refactored to make it easier to extend. Rotary-wing entity behaviors are unaffected by the changes. The API changes encompass all of the controllers, actuator, state repository and parameters associated with the rotary-wing model. The core concept behind these changes are the introduction of a separate autopilot controller, inserted between the task controllers and the rotary-wing actuator.

Previously, task controllers were directly connected to the rotary wing actuator as follows:



Now, all rotary-wing task controllers connect to a single autopilot controller, connected to the rotary-wing actuator.



Please see the following header files for details:

- ♦ Task controllers:
 - *rotWingJoyCtl.h*
 - *rotWingLandingController.h*
 - *rotWingTaskCtl.h*
 - *rotWingTurnToHeadingCtl.h*
 - *rotWngColAvod.h*
 - *rotWngEmbarkationController.h*
 - *rotWngFolwCtl.h*
 - *rotWngLoitCtl.h*
 - *rotWngMvAlg.h*
 - *rotWngMvToCtl.h*
 - *rotWngMvToLocCtl.h*
 - *rotWngPtlBtwn.h*
 - *rotWngPtrlRt.h*
 - *rotWngWaitCtl.h*
- ♦ Autopilot:
 - *rotWngPltCtl.h*
 - *simpRotWngPlt.h*
- ♦ Actuator: *rotWingAct.h*
- ♦ State Repository: *rotWingSR.h*
- ♦ Parameters: *rotWingPar.h*
- ♦ Component Descriptors:
 - *rotWngPltDesc.h*
 - *rotWngColAvDsc.h*
 - *rotWingRtTaskDesc.h*
 - *rotWingTaskDesc.h*
 - *rotWngActDesc.h*
 - *rotWngJoyDsc.h*

11.1.3. Changes to Registration for Task and Set Data Request Messages

VR-Forces 3.10 introduces some new functionality into the task manager, and a new set data manager that require some changes to the way that controllers register for task and set data request messages.

Task Messages

Any controller that registers for a task message and carries out a task must be changed to the new scheme. The change is as follows:

Old code:

```
myRadio->addRadioMessageCallback(DtRetreatTaskType,  
    retreatTaskCallback, (void *) this);
```

New code:

```
addTaskProcessorCallback(DtRetreatTaskType, retreatTaskCallback,  
    (void *) this);
```

The controller no longer calls `startTask()` when it begins processing a new task. The task manager already knows a new task is starting, and has cleaned up any previous task before the registered callback is called.

See the `addTaskSim` example for an example of a controller processing a task.

Set Data Request Messages

Any controllers that registers for a set data request message and carries out the set request must be changed to the new scheme. There is a new *DtSetDataManager* associated with each entity, which centralizes the distribution of set data requests. Code changes should be made as follows:

Old code:

```
radio->addRadioMessageCallback(DtSetHeadingRequestType,  
    setHeadingCallback, (void *) this);
```

New code:

```
entity()->setDataManager()-  
    >addSetDataProcessorCallback(DtSetHeadingRequestType,  
    setHeadingCallback, (void *) this);
```

11.1.4. Simulation API Changes

The Simulation API has the following additional changes:

- ♦ The deprecated class *DtTargetDetectionSensor* is scheduled to be removed from the API immediately following this release. See section 5.11 "Sensors" in *VR-Forces Developers Guide* for a discussion of the Signature Sensor system that functionally replaces this component.
- ♦ The load-ammo-time and unload-ammo-time parameters are no longer saved to the order of battle file. They are now strictly read in from the entity's parameter database entry. As a result, the actual times used in scenarios may be different than in previous versions. Consult the OPE files to see the actual values used.

11.1.5. GUI API Changes

The GUI API has undergone substantial changes, designed to make common extensions to the application easier to make. Among the areas of API affected by the changes:

- ♦ Capturing and Displaying State Updates
- ♦ Creating and Modifying Symbols
- ♦ Creating and Modifying Toolbars and Menus
- ♦ Drawing the Terrain Map
- ♦ Event Signaller
- ♦ Event Controllers
- ♦ Event Handlers
- ♦ Managing Overlays

For details, please see Chapter 11, *The VR-Forces GUI API*, in *VR-Forces Developers Guide*.

The GUI API has the following additional changes:

- The data member `DtTMAApplication::myModelDataDict()` (*tmApplication.h*) has changed from a stack variable to a pointer.
- The signature of the member function `DtTMWindow::init()` (*tmWindow.h*) has changed. If you have overridden this member function, be sure to change the signature in your derived version, to ensure it gets called.
- If you have created subclasses for any of the following VR-Link state repository classes in the back-end, you must change the class you inherit from as shown in [Table 11-1](#) (declared in *vrfStateReps.h* and *controlObjSR.h*):

Table 11-1: State repository classes to change

VR-Link state repository	Replace with extended VR-Forces version
<i>DtEntityStateRepository</i>	<i>DtVrfVrlEntityStateRepository</i>
<i>DtAggregateStateRepository</i>	<i>DtVrfVrlAggregateStateRepository</i>
<i>DtEnvironmentProcessRepository</i>	<i>DtControlObjectStateRepository</i>

- Subclasses of the task, set, and conditional dialog boxes that display a list box (for example, those used in the `fuelDepot`, `addTaskGui`, and `addConditionGui` examples) now take an extra parameter to the `init()` member function:

```
virtual bool init(DtTMapArea* map, const DtSimTask* baseTask = NULL, int
    iNumListBoxes = 1);
virtual bool init(DtTMapArea* map,
    const DtSetDataRequest* baseSet = NULL, int iNumListBoxes = 1);
virtual bool init(DtTMapArea* map,
    const DtSimCondExpr* baseExpr = NULL, int iNumListBoxes = 1);
```

If your subclasses override `init()` member functions, make sure to add the new parameter. You can then dynamically case the incoming object (if it is not NULL) to populate your dialog with initial information to allow users to edit your new tasks, sets or conditionals.

- The *DtTdbDrawer* has been removed and replaced with a series of classes that are used specifically for a part of terrain drawing (polygon drawer, feature drawer, and so on.) If you have previously overridden the functionality to change how grid line labels are displayed, you must subclass the *DtTerrainGridLineDrawer* rather than the *DtTdbDrawer* as follows:

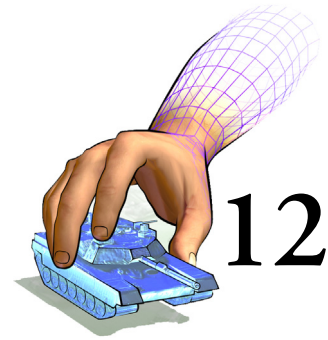
```
class MyGridLineDrawer : public DtTerrainGridLineDrawer
{
public:
    MyGridLineDrawer();
    virtual ~MyGridLineDrawer();

    static DtTerrainGridLineDrawer* create()
    { return new MyGridLineDrawer; };
};
```

And, in the main.cxx:

```
DtTacticalGdbDrawer::setTerrainGridLineDrawerCreatorFcn(MyGridLineDrawer  
::create);
```

In the subclass of the grid line drawer, the member functions to override are the same (labelForHorizLine() and labelForVertiLine().)



Migrating from 3.10 to 3.11

This chapter contains the instructions for moving from VR-Forces 3.8 to VR-Forces 3.9.

12.1. API Changes

This section describes the major changes to the VR-Forces APIs.

12.1.1. Human Movement Model

In previous releases, individual lifeforms used components derived from the automotive components. Lifeforms now use components designed to more closely simulate human movement. For details about new lifeform components, please see the class documentation.

12.1.2. Entity-Based Plan Management

In previous versions of VR-Forces, plans were managed by a central Plan Manager. The *DtSimPlanManager* has been replaced by a *DtVrfObjectPlanManager* which is attached to each object. The centralized functions of *DtSimPlanManager*, such as reading and writing plan files, have been moved to *DtVrfObjectPlanAdministrator* which is owned by *DtVrfObjectManager*. The new architecture is much cleaner because now the *DtSimTaskManager* can directly communicate with the plan manager without having to send interface messages. It's also more efficient because there is no longer any need to do lookups in a secondary object hash in *DtSimPlanManager*. For details, please see Chapter 9, [Plans](#), in *VR-Forces Developers Guide*.

12.1.3. New Communications Mechanism

In previous versions of VR-Forces, there was no distinction between messages used for internal VR-Forces, for example, a set data request message, and messages that simulated radio traffic. The communications mechanism has been changed to separate these types of messages. This change supports the new ability to send radio messages from entities to other specific entities. For details, please see Chapter 7, [Entity Communication](#) in *VR-Forces Developers Guide*.

12.1.4. Support for Plug-ins in the OPD Editor

In previous version of VR-Forces, the OPD Editor was restricted to configuring the components shipped with VR-Forces. The OPD Editor now supports the VR-Forces plug-in architecture, so that you can update it to handle components created by customers.

12.1.5. Entities Share Detection Lists

Because of the new detection levels feature (please see [Multiple Detection Levels for Discovered Targets](#)), targeting of hostile entities depends, in part, on the certainty of detection. Entities can now share their detection lists through the use of spot reports. The spot report information augments the contacts that an individual entity has made. For details, please see [Section 8.6.1, "Target Detection and Spot Reports"](#), in *VR-Forces Users Guide*.

12.1.6. Versioning of VR-Forces Messages

In the past, when different versions of VR-Forces tried to play together in a simulation, differences in message formats could cause crashes. Messages will now be tagged with the VR-Forces version so that in the future, VR-Forces can filter out messages from incompatible versions.

12.1.7. Improved API Class Documentation

The HTML class documentation now includes additional pages that provide high-level descriptions of the VR-Forces component architecture and describe how components and systems are interrelated. The new documentation also includes lists of the components used by each entity type and has links to the parameter descriptions in the OPD Editor online help. This updated documentation replaces several appendices that were formerly included in *VR-Forces Developers Guide*.

12.1.8. Upgrade to Qt 4.3.3

The VR-Forces GUI and all of its tools (OPD Editor, TDB Tool, and so on) have been migrated to Qt 3.3.5. The Qt API changed extensively from Qt 3.x to Qt 4.x. If you have modified the VR-Forces GUI and want to migrate to VR-Forces 3.11, you must upgrade to the supported version of Qt and migrate your customized code.

This section covers the main points you should consider when upgrading source code from Qt 3.x to Qt 4.x. If you do not find information in there that you need, please look at the <http://doc.trolltech.com/4.2/porting4.html> web site or email support@mak.com with your specific questions.

General Advice

When upgrading to the new API, you should check the prototype of the method you are overriding for changes. The main changes have been made to the types of container classes and constructors. If you find that a slot is not being called, a method not being called, or a constructor not being compiled, it is due to a method prototype change. In general, the compiler will catch most of the issues with upgrading.

gen.pro Changes

The new qmake system supports single pass project file generation, so, there is no longer a need for a separate merge program. This means, however, that the "debug" and "release" (or !debug) configuration options have changed to DebugBuild and ReleaseBuild. If you have either of these tags in your *.pro* file, change them accordingly. The supplied project generation system has been updated to reflect these changes.

Example Project Files

In previous installations, the project files for examples were pre-generated. With Qt 4.X, this has been changed. The QTDIR environment variable has been removed in favor of hard coding the path to the Qt installation. Because of this change, it is now necessary to generate the example project files when you want to build them. On Windows, make sure the qmake application can be found in your Windows path and then run the "generateExamples" batch script from the examples directory. Once run, you can open the examples solution to build the example projects.

The exceptions to this are the back-end examples and terrain toolkit examples. Since the back-end and terrain toolkit examples do not need Qt in order to be built or run, those project files are distributed.

Constructors

- ♦ The "name" parameter has been removed from all Qt based objects. If you need to name the object class that you are creating for identification purposes, in your constructor use the setObjectName() method to set the name. You can use the name passed into the old constructor, however, the name is not passed to the parent object.

- ♦ The constructor for a popup menu has changed from:

```
MyPopupMenu(QObject* parent, const char* name);
```

to:

```
MyPopupMenu(QWidget* parent, bool display = true);
```

UI Files

All *.ui* files must be converted using the `uic3` program (in `Qt_installdir/bin`). This program, while helpful, does not complete the conversion process. After the conversion you must inspect the resultant *.ui* file for any object that references Q3 (for example, `Q3GroupBox`). If such objects are present, hand edit them to remove the Q3 nomenclature. In many cases, it might be easier to simply recreate your *.ui* file using the new Qt designer for 4.X.

Classes generated by the new *.ui* files cannot be subclassed. If your class used to look like this:

```
class MyWidget : public MyWidgetFromUi
{
    ....
};
```

it must now have the *.ui* generated class as a member variable, like this:

```
class MyWidget : public QWidget
{
public:
    MyWidget(QWidget* parent, const Qt::WindowFlags& flag);
public:
    Ui::MyWidgetFromUi myUi;
};

MyWidget(QWidget* parent, const Qt::WindowFlags& flags) :
    QWidget(parent, flags)
{
    myUi.setupUi(this);
}
```

Container Classes

The container classes have been completely changed for Qt 4.X to resemble standard template classes. Most container classes have been converted to either a `QVector`, a `QList`, or a `QHash` class. The major container changes are:

- `QPtrVector<type>` and `QValueList<type>` are now `QVector` (for example, `QValueList<DtPoint>` and `QPtrVector<DtPoint>` is now `QVector<DtPoint>`).
- `QDict<type>` is now `QHash<QString, type>`
- `QIntDict<type>` is now `QHash<int, type>`
- `QPtrDict<type>` is now specific to a particular pointer type. For example, if you create a dictionary based on a `MyObject`, the declaration looks like:
`QHash<MyObject*, type>`

Menu Changes

The menu system API has changed. We have created a system similar to 3.X, to minimize the amount of migration work needed. To support this, the following changes have been made:

- ♦ The `initMenuItem` (and `initToggleMenuItem`) methods used to take two parameters when a menu item was added. Now they only take a single text parameter. So, if your init call looked like this:

```
insertMenuItem(DtNoMenuItem, DtFilterForceTypeAction,  
    trUtf8( "Force Type" ), trUtf8( "Force Type" ),  
    trUtf8( "Force Type Filter." ));
```

it should now look like this:

```
insertMenuItem(DtNoMenuItem, DtFilterForceTypeAction,  
    trUtf8( "Force Type Filter." ));
```

- ♦ `QActionGroup` is now `DtActionGroup`. If you create an action group in `initActionGroups()`, make sure to create a `DtActionGroup` class. If you return an action group in `getActionGroupForType`, make sure to change the prototype to return a `DtActionGroup`. When creating an action group, if your code looked like:

```
myFilterActionGroup = new QActionGroup(parent(), "myFilterActionGroup");
```

it should now look like:

```
myFilterActionGroup = new DtActionGroup(this);
```

If you want your action group to be an exclusive check area, make sure to call `setExclusive(true)` on the action group (or `false` if you do not want it to be exclusive).

`DtModelKey` is now a `QString` and is no longer its own class. If you have code which looks like:

```
data->key().string()
```

you can remove `.string()`.

The constant `WFlags` has been changed to `Qt::WindowFlags`.

12.1.9. Miscellaneous API Changes

- ♦ *DtTargetDetectionSensor*, *DtTargetDetectionPSR*, and *DtTargetDetectionSensorDescriptor* have been removed. They were replaced in VR-Forces 3.9 by the object sensor model. For details, please see Chapter 5, [Creating New Components](#), in *VR-Forces Developers Guide*.
- ♦ The *DtFixedWingPilotControllerDescriptor* now has **at-time** and **near-time** parameters, which used to be hard-coded in the controller's `atGoal()` and `nearGoal()` functions.
- ♦ `DtSimTaskManager::removeAllReferancesToController()` was misspelled. It has been renamed to `removeAllReferencestoController()`.
- ♦ `DtVrfObjectManager::defineCustomeStateRepositories()` and the *DtVrfVrEntityStateRepository* class it created have been removed; the derived state repository provided a work-around for a VR-Link bug which has since been fixed.
- ♦ A new front-end class, *DtVrfGuiDisplayOptions* has been introduced. If you have (VRF only) subclassed options from *DtPvdDisplayOptions*, you must now subclass from *DtVrfGuiDisplayOptions*. If you do not, unpredictable results may occur.
- ♦ Many elements that were deprecated in previous releases have been removed from the GUI and simulation APIs.
- ♦ The `windowsInfoExtension` example has been renamed `addGuiCoordinateSystem`.
- ♦ The `viewControllerObjects` example has been renamed `viewFilter`.
- ♦ The `rangeFilter` example was renamed to `vrfGuiRangeFilter`



Link - Simulate - Visualize

VRF-3.12-9-090601