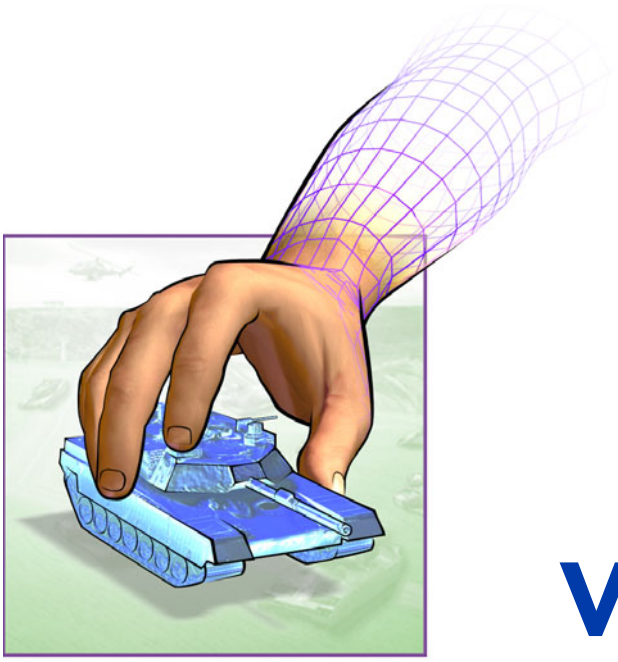
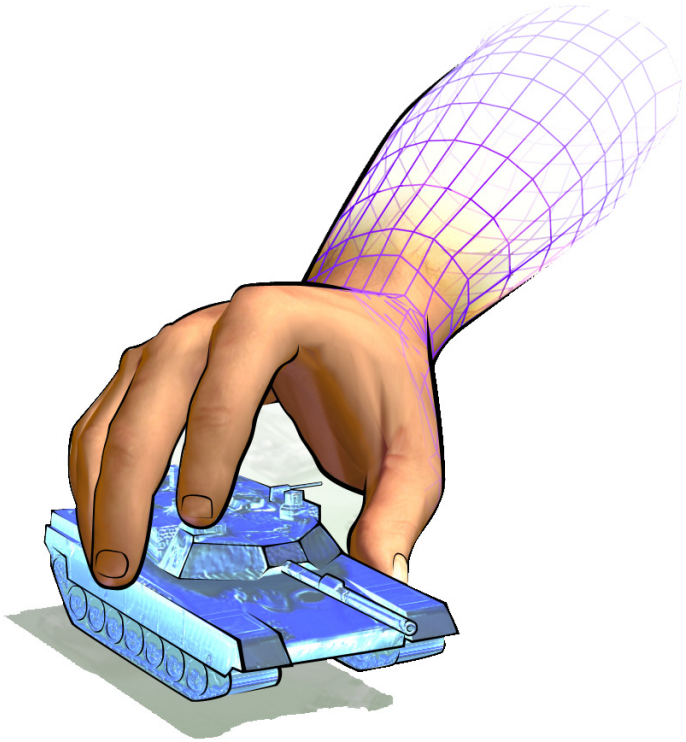


VR-Forces

Migration Guide

VT MÄK
A company of VT Systems



VR-Forces

Migration Guide

VT MÄK
A company of VT Systems

For third-party license information, please see "Third Party Licenses," on page x.

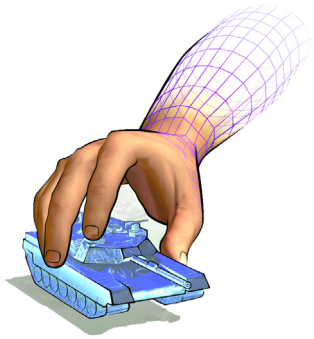
VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

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

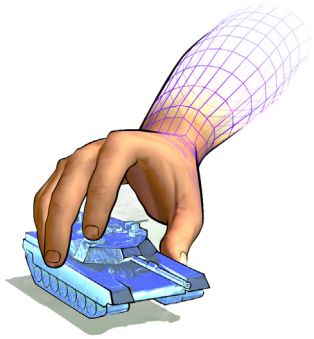
info@mak.com

www.mak.com

Revision VRF-4.0.4-9-120608



Contents



Preface

This manual is for developers who must migrate applications from VR-Forces 3.12 to VR-Forces 4.0.x.

VR-Forces Documentation

VR-Forces documentation is provided as manuals in PDF format, online help, and HTML class documentation. The PDF files are in the *./doc* directory. The VR-Forces documentation set is as follows:

- *VR-Forces Documentation Center* is your central starting point for accessing the VR-Forces manual set. It has a documentation roadmap, tables of contents for all manuals, and a master index, including the 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. All table of contents entries and index entries are live links to the manuals.
- *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 VR-Forces and configure license management. It explains how to use the VR-Forces graphical user interface to view simulations and how to manage aspects of VR-Forces that are not directly related to creating and running scenarios.
- *VR-Forces Scenario Management Guide* explains how to create and run scenarios.
- *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.
- *VR-Forces Developers Guide* explains how to use the VR-Forces simulation and remote control APIs. It focuses primarily on the simulation engine.

- *VR-Forces Front-End Developers Guide* explains how to extend or modify the VR-Forces graphical user interface (GUI).
- *VR-Forces Migration Guide* collates API migration information for each release in the VR-Forces 3.x series.
- Online help. The VR-Forces front-end, the OPD Editor, the Entity Editor, and the TDB Tool have online help accessible from the Help menu.
- Class documentation. Classes are documented in linked HTML pages.
- *VR-Forces Release Notes*.

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) and the Distributed Interactive Simulation (DIS) protocol. VR-Link has built-in support for the RPR FOM and allows you to map to other FOMs. This library minimizes the time and effort required to build and maintain new HLA or DIS-compliant applications, and to integrate such compliance into existing applications.

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

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

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

- VR-Vantage™. VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes four end-user applications (VR-Vantage Stealth, VR-Vantage XR, VR-Vantage PVD, and VR-Vantage IG), the VR-Vantage Toolkit, and VR-Vantage FreeView.
  - VR-Vantage 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.
  - VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.
  - VR-Vantage XR provides a common operational picture using the same 3D views as VR-Vantage Stealth, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world.
  - VR-Vantage PVD provides a 2D plan view display. It gives you the big picture of the simulated world.
  - The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.
  - VR-Vantage Free View is a terrain and 3D model viewer that introduces some of the features of VR-Vantage applications in a useful, free utility.
- MÄK Data Logger. The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.

- VR-inTerra. VR-inTerra is a C++ API for adding terrain agility to applications. It can load, page, or stream terrain from a wide variety of formats or sources into a single, consistent run-time representation, consisting of a collision graph and vector network.

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

Post

Send postal correspondence to:	VT MÄK 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.
`Monospaced Italic`	Indicates command variables that you replace with appropriate values.
Blue text	A hypertext link to another location in this manual or another manual in the documentation set.
Blue bold text	A hypertext link to class documentation.
`{ }`	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.
Italic	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**.
	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*.

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 web site is: www.boost.org.

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

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

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

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

libXML and libICONV

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

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

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:

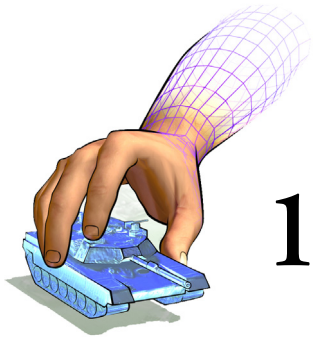
- EHS: http://xaxxon.slackworks.com/ehs/
- PCRE: http://www.pcre.org/
- PME: http://xaxxon.slackworks.com/pme/index.html

LizardTech

Portions of this computer program are copyright © 1995-2010 Celartem, Inc., doing business as LizardTech. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Freefont OpenType Font Set

VR-Vantage applications and VR-Forces use the Freefont OpenType font set from the Free Software Foundation. It is covered by the General Public License (GPL). For details, please see: http://www.gnu.org/licenses/gpl.html



1

Introduction

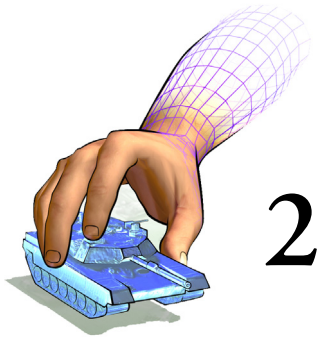
This chapter provides a brief introduction to the contents of this guide.

1.1. Migrating from VR-Forces 3.x to 4.x

Before migrating from VR-Forces 3.12 to 4.x, you should understand the major areas that have changed:

- The front-end was completely redesigned. In VR-Forces 3.12, the 2D and 3D interfaces were separate executables. VR-Forces 4.x has a completely integrated 2D/3D graphical user interface (GUI) based on the VR-Vantage toolkit. As a consequence, the GUI API has changed dramatically.
- The Simulation API and Remote Control API have incremental improvements to support new features, but are largely similar to previous releases.
- The Terrain API has been updated to support the terrain agility features used in VR-Vantage and to support synchronization of the front-end and back-end views of the terrain.

This migration guide provides a high-level review of changes to the major classes in the GUI API. However, it is not a comprehensive guide to changes. MÄK has decided not to release a comprehensive porting guide because we believe the upgrade experience is likely to be highly specific to individual customer use cases. As such, a comprehensive porting guide would be unlikely to fully explain the process no matter how much we wrote. Instead, MÄK has pledged to work with customers on a one-on-one basis during their upgrade process. That will include help through technical support, customer specific webinars and possibly on-site support and training. Please contact us a support@mak.com or through your MÄK salesperson or reseller for help with porting to VR-Forces 4.0.



2

Migrating from VR-Forces 3.12 to 4.0.3

The VR-Forces 4.x GUI is built with the VR-Vantage Toolkit, which is significantly different from the VR-Forces 3.12 GUI API. This migration guide describes the major differences between VR-Forces 3.12 and VR-Forces 4.0 through 4.0.3.

2.1. Getting Information About Objects

In VR-Forces 3.12 *DtModelKey* was used to find object information in various dictionaries, most notable the *DtModelDataDictionary*. In VR-Forces 4.x this has been replaced with *DtElementId*. The *DtElementId* is the main key to use to look up data in the system. For details, please see "Storing Data," on page 2-2.

The API extends the element ID with the scene object ID. The scene object ID (*DtUniqueId*) represents an individual screen representation of an object, given a particular model set. For example, the 2D representation of an object has a different scene object ID than the 3D representation. If you have a scene object ID, you can use the *DtElementData* object to retrieve the element ID based on scene object ID. If you need a scene object ID from an element ID, the *DtElementData* can also do that mapping.

Selection sets (discussed in "Managing Object Selection," on page 2-3) contain the *DtElementId* of the objects selected. You can use these IDs to retrieve information about selected objects.

2.2. Storing Data

In VR-Forces 3.12, *DtModelData* was used as the foundation for data storage, symbol creation, and symbol updating. The *DtModelData* class does not exist in VR-Forces 4.x. It has been replaced with the *DtStateView*. A *DtStateView* is a template class that takes a *DtSimState* object as a template. This object is implemented as a subclass, and those subclasses contains the data that represents the information about the object. In VR-Forces 4.x, the main *DtSimState* objects are:

- *DtVrlinkSimulatedBaseState*. Information about any VR-Link-based object, such as marking text, type, force
- *DtVrlinkSimulatedEntityState*. Information about VR-Link entities.
- *DtVrlinkSimulatedAggregateState*. Information about VR-Link aggregates.
- *DtVrlinkSimulatedEnvironmentProcess*. Information about VR-Link environmentals.
- *DtVrfObjectDataState*. Information about an object that is VR-Forces-specific

When an object is discovered on the network, information about that object from its VR-Link reflected object is placed into the appropriate object state. This state is added to a queue for updating.

In VR-Forces 3.12, as soon as an object update was received over the network, it was immediately updated. In VR-Forces 4.x this is not the case. Each object is updated at a certain frame rate. This allows for better performance and control over when object information is updated. For example, if an object is selected, its object update rate is increased, assuming the user might want to know more information about this object. This means all other objects continue to update at a default rate, or not at all. This does not effect the position, speed, or orientation of the object, simply the additional data that is transmitted to the front-end for that object.

This information is stored in *DtStateViewCollection* classes. Each *DtStateViewCollection* keeps a list of all particular objects of a state view type and can be used to retrieve information about that object type. So, for example, there are collections of entities, aggregates, and environmental processes.

In VR-Forces the *DtVrfStateViewCollectionManager* manages all these collections. It is your way of retrieving information about a particular object and its *DtStateView* information. The *DtVrfStateViewCollectionManager* is analogous to the VR-Forces 3.12 *DtModelDataDictionary* class.

In the *DtVrfStateViewCollectionManager*, all objects that are in the system are stored in the mySimulatedBaseStateViewCollection member. This member holds all structures that are of a *DtVrlinkSimulatedBaseState* (which all state view objects subclass from). Using the findStateView() methods of the *DtVrfStateViewCollectionManager*, you can find the basic state view information about any object. This information can be retrieved by marking text name or by *DtElementId* of the object. If you know that an item is specifically an entity, aggregate, or environmental, you can use the state view collection that is appropriate to find the state data for that object.

2.3. Managing Object Selection

In VR-Forces 3.12 you would use the map area of the currently active window to get a list of objects that were currently selected. In VR-Forces 4.x you use the *DtSelectionManager* class to get the list of object selected, as follows:

```
DtSelectionManager::IdSet currentSelections =
  DtSelectionManager::instance(myDe).currentSelection();
```

You can then iterate over the list to get information about these objects from the *DtVrfStateViewCollectionManager*:

```
DtSelectionManager::IdSet::const_iterator iter =
  currentSelections.begin();

while (iter != currentSelections.end())
{
  DtStateView<DtVrlinkSimulatedEntityState>* stateView =
    const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
    DtVrfStateViewCollectionManager::instance(myDe).entityStateView()
    ->findStateView( *iter ) );
  ++iter;
}
```

VR-Forces 4.x also supplies a convenience class called the *DtVrfSelectionHandler*. This class has methods for retrieving information about current selections. For example, if you want to get data for all the currently selected objects, you can use the code in the previous example or you could do the following:

```
std::vector<makVrv::DtSimEntry*> entries =
  DtVrfSelectionHandler::instance(myDe).getSelectedEntries();
```

Remember that each *DtSimEntry* contains the list of all available data for that object. Most objects in VR-Forces have two *DtSimEntry* views of data: the VR-Link State Repository data (*DtVrlinkSimulatedEntityState*) and the VR-Forces-specific object data (*DtVrfObjectDataState*). You can iterate over the simStates() of the state entry, dynamically casting to the object in question that you want. Once the dynamic cast succeeds, you can use that data.

2.4. Creating Symbols

In VR-Forces 3.12 if you wanted to affect the way that symbols were created, you would override the *DtMtlSymbolMapper*, and install your own version.

In VR-Forces 4.x, symbols are created from *DtVisualDefinitionSets*. These visualizer sets are organized in *DtObjectDictionary* classes. Each class of object (entity, aggregate, environmental, interaction) has its own object dictionary. In order to affect the look of a symbol, you can modify the object dictionary, adding your own *DtVisualDefinition* or *DtVisualDefinitionSet*. The exampleObjectDictionary, exampleRangeLine and exampleVisualDefinitionSet examples show how this is done.

Mapping entity type enumerations to models is now done in the Entity Type Mappings dialog box rather than in symbol map files. Entity types are mapped to entity definitions that contain the correct visualizer information for that entity type.

In VR-Forces 4.x, the actual visual definitions are not created in the main GUI thread. A *DtElement* object (or a subclass) is created. It contains all the visualizers needed to visualize an object. *DtElement* objects are created in the network thread and are protected from the main thread. If you want to, in code, change how these items look, you will need to do that before creation by changing the visual definition set for that particular object. Individual visualizers, however, can be created to self configure given certain conditions. The point being stressed here is that there has been an attempt to move away from direct manipulation of the symbols and move towards a more object oriented approach for symbol manipulation.

If you need to get access to an individual symbol, you can do this through the *DtAgentManager*. Since each object is scene dependent, you will need to know the model set you are referencing to get the scene object ID of the object you care about:

```
DtElementData::SceneObjectIdList ids;

myDe.dataBank().elementData().getSceneObjectIds(elementId, ids,
  myDe.driverManager().inputDriver().
  currentChannel()->currentModelSet());
DtUniqueID idToUse = elementId;

if (ids.size())
{
  idToUse = ids[0];
}

DtAgentUpdateResolverInterface* updater =
  myDe.agentManager().findUpdater( idToUse );
```

```
if(updater)
{
  DtSceneObject* sceneObject =
    updater->castObjectTypeFromUpdater<DtSceneObject>( "DtSceneObject");

  if ( sceneObject )
  {
    sceneObject->getPosition(0, dbLocation);
  }
}
```

Please refer to the *DtSceneObject* class for the information you can retrieve.

The scene objects are screen representations only and do not contain simulation data.

2.5. Updating Symbols

In VR-Forces 3.12, in order to affect how a symbol was updated, you would have either subclassed and installed your own version of the *DtVrfGuiSymbolUpdater* class or added post update callbacks.

In VR-Forces 4.x there is no centralized symbol updater. Each visualizer is responsible for doing its own updating. When a visualizer is created, it is handed a subclass of a *DtStateListener*. This class contains all the simulation data necessary to drive the visuals of an object. Signals are defined in these state listener subclasses that visualizers can connect to in order to update their state.

If you want to change how an existing object is updating, you must subclass and install that object into the *DtVrlinkStateVisualizerFactory* if this object has a VR-Link representation, or the *DtStateVisualizerFactory* if it does not.

To add subclasses to the VR-Link state visualizer factory you must create driver accessories for the VR-Link protocol you want to change. (Please see the addAttr example for an example of how to create an accessory.) You can then reference the VR-Link state visualizer factory as in the slot_onSimCreated() method (in *DtAddAttrGuiAccessory.inl*):

```
DtVrlinkStateVisualizerFactory& visualizerStateFactory =
  DtVrlinkStateVisualizerFactory::instance(
  (DtVrlinkConnection*)connection );
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

The same methodology can be used for the state visualizer factory:

```
DtWriteSettingsLock<DtSharedStateVisualizerFactory>
  visualizerStateFactory(DtSharedSettingsManager::instance(myDe));
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

2.6. Sending Messages to the Back-end

In VR-Forces 3.12, to communicate with the back-end (sending sets, tasks or interface content messages), the remote controller was used: DtVrfGuiAppEventController::remoteController().

In VR-Forces 4.x there is no direct access to the remote controller from the GUI thread. Since the GUI is not network dependent, the *DtVrfSimMessager* class was created to interface the GUI to the back-end. This class should be used to send all messages. The interface content example shows how to send interface content messages using *DtVrfSimMessager*.

2.7. Adding Callbacks to Receive Messages

In VR-Forces 3.12 the *DtVrfMessageInterface* class was used to register callback messages for message types. In VR-Forces 4.x the *DtVrfSimMessageHandler* is used. The *DtVrfSimMessager* also interfaces with this class, and that can be used as well. The interface content example shows how to register for a callback message using *DtVrfSimMessageHandler*.

2.8. Keyboard and Mouse Handling

In VR-Forces 3.12 you would subclass and install an event handler to handle mouse events and keyboard events. You might also have subclassed and installed your own version of the *DtPvdMapArea* or connected to signals to handle events.

In VR-Forces 4.x you subclass and install a *DtEventProcessor*. The *DtEventProcessor* subclasses have methods for mouse and keyboard messages. If you want to handle a mouse or keyboard message, override the appropriate method (processKeyboardEvent() or processMouseEvent()). If you handle the message you can return true or false. You can also be notified of events that were handled by other event processors before your event processor got a chance, by overriding the above methods and handling the case of a MouseLost event.

Add your event processor subclass into the *DtInputDriver* class. The *DtInputDriver* object is referenced through the driver manager:

```
DtInputDriver& inputDriver = myDe.driverManager().inputDriver();
```

The event processor can be added to the front or the back of the list (addEventProcessorToFront() or addEventProcessorToBack()). You can also get the list of event processors and insert it directly (eventProcessorList()). The exampleEventProcessor example has an example of how to use an event processor.

2.9. Changes to Signal Usage

VR-Forces 3.12 mostly used Qt signals. VR-Forces 4.x mostly uses Boost signals. While some Qt signals are used in the Qt layer, they are usually translated back into Boost signals.

To use a Boost signal you must connect to a signal and give a method to call when the signal is invoked.

The following examples show how to connect to signals without parameters (preTick), and with parameters (postTick). The binding is done to an object class and an instance of that object.

```
myDe.signal_preTick.connect(boost::bind(&MyObject::method, this));
myDe.signal_postTick.connect(boost::bind(&MyObject::method, this, _1));
```


It is very important to disconnect signals when they are no longer needed or when the object is freed. Failing to disconnect signals can lead to application instability, as, unlike Qt signals, when the object is destroyed the signal is not disconnected.

2.9.1. Application Tick Notification

In VR-Forces 3.12 there was no way to be notified when the application was about to tick or had completed a tick. In VR-Forces 4.x, you can get this information. The main class of a VR-Vantage application is *DtDe*. The *DtDe* class is typically passed to any object that might need to use it, and is available to any plug-in on startup.

To be notified when the application is about to tick, connect to the signal_preTick signal.

On a post tick, connect to the signal_postTick signal.

2.9.2. Selection Management

In VR-Forces 3.12, to find out when selections had changed, you would connect to the Qt signal sent by the map area on selection changed (selectionUpdated, selection-Changed, and so on). In VR-Forces 4.x, you use the Selection Manager and its signals to be notified when selections have changed. You can get a reference to the Selection Manager as follows:

```
DtSelectionManager& selManager = DtSelectionManager::instance(myDe);
```

The signal_currentSelectionChanged signal is sent any time the current selection is changed. It supplies the following parameters:

- The current selection.
- What was unselected.
- The new selection type.
- The old selection type.

The IDs supplied are element IDs, which can be used to retrieve data about the selected objects. "Managing Object Selection," on page 2-3 explains how to use this information.

The signal_vertexSelected and signal_vertexDeselected signals are sent when vertices on a line are selected or unselected

2.9.3. Tracking New Objects

In VR-Forces 3.12, to find out when an object was added, you would have used the modelDataAdded signal. In VR-Forces 4.x you use the *DtElementData* object. You can get to the *DtElementData* as follows:

```
DtElementData& elementData = myDe.dataBank().elementData();
```

The signal signal_elementsAdded is sent for any elements added during that tick.

You can then use the State View Collection Manager to find information about the elements added. Due to the nature of the multi-threading of the application, it is possible that the state view data does not yet exist for those elements. In that case, for the state view collection you want to retrieve information for, connect to the signal_stateViewAdded signal. For example:

```
DtVrfStateViewCollectionManager::instance(myDe).
  baseStateView()->signal_stateViewAdded
```

When the state view is added you can then get information about that object.

2.9.4. Tracking Object Updates

In VR-Forces 3.12, to be notified when an object was updated, you might have installed your own version of the symbol updater or connected to the model data dictionary's modelDataUpdated signal. In VR-Forces 4.x you use the state view itself to be notified when state view data is updated.

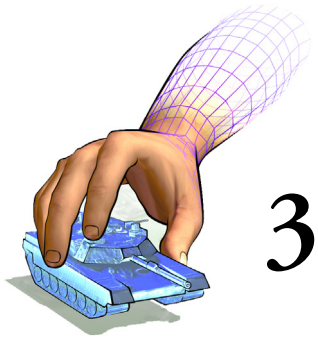
You first need to find the state view you are interested in:

```
DtStateView<DtVrlinkSimulatedEntityState>* stateView =
  const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
  selHandler.entityStateView()->findStateView( principalSimEntryId ) );
stateView->signal_stateViewUpdated.connect( boost::bind(
  &MyObject::slot_onPrincipalStateViewUpdated, this, _1));
```

You will now be notified when that object is updated. This is more efficient than the way it was done in VR-Forces 3.12, because now you can be notified when a very specific instance of the data you are interested in is updated.

2.9.5. Tracking Object Removals

In VR-Forces 3.12 you would have used the modelDataAboutToBeRemoved signal. In VR-Forces 4.x you use the element data's signal_elementsToBeRemoved signal. This provides a list of elements that are being removed in this tick. Please see "Tracking New Objects," on page 2-8 for how to retrieve a reference to the *DtElementData*.



3

Migrating from VR-Forces 3.12 to 4.0.4

The VR-Forces 4.x GUI is built with the VR-Vantage Toolkit, which is significantly different from the VR-Forces 3.12 GUI API. This migration guide describes the major differences between VR-Forces 3.12 and VR-Forces 4.0.4.

3.1. Getting Information About Objects

In VR-Forces 3.12 *DtModelKey* was used to find object information in various dictionaries, most notable the *DtModelDataDictionary*. In VR-Forces 4.x this has been replaced with *DtElementId*. The *DtElementId* is the main key to use to look up data in the system. For details, please see "Storing Data," on page 3-2.

The API extends the element ID with the scene object ID. The scene object ID (*DtUniqueId*) represents an individual screen representation of an object, given a particular model set. For example, the 2D representation of an object has a different scene object ID than the 3D representation. If you have a scene object ID, you can use the *DtElementData* object to retrieve the element ID based on scene object ID. If you need a scene object ID from an element ID, the *DtElementData* can also do that mapping.

Selection sets (discussed in "Managing Object Selection," on page 3-3) contain the *DtElementId* of the objects selected. You can use these IDs to retrieve information about selected objects.

3.2. Storing Data

In VR-Forces 3.12, *DtModelData* was used as the foundation for data storage, symbol creation, and symbol updating. The *DtModelData* class does not exist in VR-Forces 4.x. It has been replaced with *DtObjectDataInterface*. This class (and subclasses) let you access data for any simulated object in the system:

- *DtEntityDataInterface*
- *DtAggregateDataInterface*
- *DtTacticalGraphicDataInterface*
- *DtSpotReportDataInterface*.

To retrieve information about objects, you can either use the object's element ID or the object's name (marking text) from the *DtVrfStateViewCollectionManager*. The *DtVrfStateViewCollectionManager* class is analogous to the *DtModelDataDictionary* in VR-Forces 3.12. In the *DtVrfStateViewCollectionManager* class you will use the lookupAndCastObjectData() method to return a shared data pointer to the data you need. For example, if you want to find an entity data:

```
DtEntityDataInterfacePtr data =
  DtVrfStateViewCollectionManager::instance(myDe).lookupAndCastObjectDat
  a<DtEntityDataInterface>("M1A2 1");
```

Also, the IDs in the current selection set can be used to look up data for selected objects.

In VR-Forces 3.12, as soon as an object update was received over the network, it was immediately updated. In VR-Forces 4.x this is not the case. Each object is updated at a certain frame rate. This allows for better performance and control over when object information is updated. For example, if an object is selected, its object update rate is increased, assuming the user might want to know more information about this object. This means all other objects continue to update at a default rate, or not at all. This does not effect the position, speed, or orientation of the object, simply the additional data that is transmitted to the front-end for that object.

This information is stored in *DtStateViewCollection* classes. Each *DtStateViewCollection* keeps a list of all particular objects of a state view type and can be used to retrieve information about that object type. So, for example, there are collections of entities, aggregates, and environmental processes.

In VR-Forces the *DtVrfStateViewCollectionManager* manages all these collections. It is your way of retrieving information about a particular object and its *DtStateView* information. The *DtVrfStateViewCollectionManager* is analogous to the VR-Forces 3.12 *DtModelDataDictionary* class.

In the *DtVrfStateViewCollectionManager*, all objects that are in the system are stored in the mySimulatedBaseStateViewCollection member. This member holds all structures that are of a *DtVrlinkSimulatedBaseState* (which all state view objects subclass from). Using the findStateView() methods of the *DtVrfStateViewCollectionManager*, you can find the basic state view information about any object. This information can be retrieved by marking text name or by *DtElementId* of the object. If you know that an item is specifically an entity, aggregate, or environmental, you can use the state view collection that is appropriate to find the state data for that object.

The guiObjectDataInterface example details how to use these data interface classes.

3.3. Managing Object Selection

In VR-Forces 3.12 you would use the map area of the currently active window to get a list of objects that were currently selected. In VR-Forces 4.x you use the *DtSelectionManager* class to get the list of object selected, as follows:

```
DtSelectionManager::IdSet currentSelections =
  DtSelectionManager::instance(myDe).currentSelection();
```

You can then iterate over the list to get information about these objects from the *DtVrfStateViewCollectionManager*:

```
DtSelectionManager::IdSet::const_iterator iter =
  currentSelections.begin();

while (iter != currentSelections.end())
{
  DtStateView<DtVrlinkSimulatedEntityState>* stateView =
    const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
    DtVrfStateViewCollectionManager::instance(myDe).entityStateView()
    ->findStateView( *iter ) );
  ++iter;
```

```
}
```

VR-Forces 4.x also supplies a convenience class called the *DtVrfSelectionHandler*. This class has methods for retrieving information about current selections. For example, if you want to get data for all the currently selected objects, you can use the code in the previous example or you could do the following:

```
std::vector<makVrv::DtSimEntry*> entries =
  DtVrfSelectionHandler::instance(myDe).getSelectedEntries();
```

Remember that each *DtSimEntry* contains the list of all available data for that object. Most objects in VR-Forces have two *DtSimEntry* views of data: the VR-Link State Repository data (*DtVrlinkSimulatedEntityState*) and the VR-Forces-specific object data (*DtVrfObjectDataState*). You can iterate over the simStates() of the state entry, dynamically casting to the object in question that you want. Once the dynamic cast succeeds, you can use that data.

3.4. Creating Symbols

In VR-Forces 3.12 if you wanted to affect the way that symbols were created, you would override the *DtMtlSymbolMapper*, and install your own version.

In VR-Forces 4.x, symbols are created from *DtVisualDefinitionSets*. These visualizer sets are organized in *DtObjectDictionary* classes. Each class of object (entity, aggregate, environmental, interaction) has its own object dictionary. In order to affect the look of a symbol, you can modify the object dictionary, adding your own *DtVisualDefinition* or *DtVisualDefinitionSet*. The exampleObjectDictionary, exampleRangeLine and exampleVisualDefinitionSet examples show how this is done.

Mapping entity type enumerations to models is now done in the Entity Type Mappings dialog box rather than in symbol map files. Entity types are mapped to entity definitions that contain the correct visualizer information for that entity type.

In VR-Forces 4.x, the actual visual definitions are not created in the main GUI thread. A *DtElement* object (or a subclass) is created. It contains all the visualizers needed to visualize an object. *DtElement* objects are created in the network thread and are protected from the main thread. If you want to, in code, change how these items look, you will need to do that before creation by changing the visual definition set for that particular object. Individual visualizers, however, can be created to self configure given certain conditions. The point being stressed here is that there has been an attempt to move away from direct manipulation of the symbols and move towards a more object oriented approach for symbol manipulation.

If you need to get access to an individual symbol, you can do this through the *DtAgentManager*. Since each object is scene dependent, you will need to know the model set you are referencing to get the scene object ID of the object you care about:

```
DtElementData::SceneObjectIdList ids;

myDe.dataBank().elementData().getSceneObjectIds(elementId, ids,
  myDe.driverManager().inputDriver().
  currentChannel()->currentModelSet());
```

```
DtUniqueID idToUse = elementId;

if (ids.size())
{
  idToUse = ids[0];
}

DtAgentUpdateResolverInterface* updater =
  myDe.agentManager().findUpdater( idToUse );

if(updater)
{
  DtSceneObject* sceneObject =
    updater->castObjectTypeFromUpdater<DtSceneObject>( "DtSceneObject");

  if ( sceneObject )
  {
    sceneObject->getPosition(0, dbLocation);
  }
}
```

Please refer to the *DtSceneObject* class for the information you can retrieve.

The scene objects are screen representations only and do not contain simulation data.

3.4.1. Creating Local (Unpublished) Symbols

In VR-Forces 3.12 you could create unpublished symbols by creating an appropriate *DtModelData* subclass and adding that model data into the application's Model Data Dictionary. You would then update that model data and mark it for update in the application to change the symbol.

In VR-Forces 4.x, there is now a *DtLocalObjectManager* class that does this for you. The guiLocalCreateObject example shows how to use this new class.

3.5. Updating Symbols

In VR-Forces 3.12, in order to affect how a symbol was updated, you would have either subclassed and installed your own version of the *DtVrfGuiSymbolUpdater* class or added post update callbacks.

In VR-Forces 4.x there is no centralized symbol updater. Each visualizer is responsible for doing its own updating. When a visualizer is created, it is handed a subclass of a *DtStateListener*. This class contains all the simulation data necessary to drive the visuals of an object. Signals are defined in these state listener subclasses that visualizers can connect to in order to update their state.

If you want to change how an existing object is updating, you must subclass and install that object into the *DtVrlinkStateVisualizerFactory* if this object has a VR-Link representation, or the *DtStateVisualizerFactory* if it does not.

To add subclasses to the VR-Link state visualizer factory you must create driver accessories for the VR-Link protocol you want to change. (Please see the addAttr example for an example of how to create an accessory.) You can then reference the VR-Link state visualizer factory as in the slot_onSimCreated() method (in *DtAddAttrGuiAccessory.inl*):

```
DtVrlinkStateVisualizerFactory& visualizerStateFactory =
  DtVrlinkStateVisualizerFactory::instance(
  (DtVrlinkConnection*)connection );
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

The same methodology can be used for the state visualizer factory:

```
DtWriteSettingsLock<DtSharedStateVisualizerFactory>
  visualizerStateFactory(DtSharedSettingsManager::instance(myDe));
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

3.6. Sending Messages to the Back-end

In VR-Forces 3.12, to communicate with the back-end (sending sets, tasks or interface content messages), the remote controller was used: DtVrfGuiAppEventController::remoteController().

In VR-Forces 4.x there is no direct access to the remote controller from the GUI thread. Since the GUI is not network dependent, the *DtGuiThreadVrfRemoteController* class interfaces the GUI to the back-end. Use this class to send all messages. The interface content example and the guiThreadVrfRemoteController example show how to send interface content messages using *DtGuiThreadVrfRemoteController*.

3.7. Adding Callbacks to Receive Messages

In VR-Forces 3.12 the *DtVrfMessageInterface* class was used to register callback messages for message types. In VR-Forces 4.x the *DtNetworkMessageCallbackManager* class is used. The *DtGuiThreadVrfRemoteController* also interfaces with this class, and that can be used as well. The interface content example and the networkCallbackManager example show how to register for a callback message using *DtNetworkMessageCallbackManager*.

3.8. Keyboard and Mouse Handling

In VR-Forces 3.12 you would subclass and install an event handler to handle mouse events and keyboard events. You might also have subclassed and installed your own version of the *DtPvdMapArea* or connected to signals to handle events.

In VR-Forces 4.x you subclass and install a *DtEventProcessor*. The *DtEventProcessor* subclasses have methods for mouse and keyboard messages. If you want to handle a mouse or keyboard message, override the appropriate method (processKeyboardEvent() or processMouseEvent()). If you handle the message you can return true or false. You can also be notified of events that were handled by other event processors before your event processor got a chance, by overriding the above methods and handling the case of a MouseLost event.

Add your event processor subclass into the *DtInputDriver* class. The *DtInputDriver* object is referenced through the driver manager:

```
DtInputDriver& inputDriver = myDe.driverManager().inputDriver();
```

The event processor can be added to the front or the back of the list (addEventProcessorToFront() or addEventProcessorToBack()). You can also get the list of event processors and insert it directly (eventProcessorList()). The exampleEventProcessor example has an example of how to use an event processor.

3.9. Changes to Signal Usage

VR-Forces 3.12 mostly used Qt signals. VR-Forces 4.x mostly uses Boost signals. While some Qt signals are used in the Qt layer, they are usually translated back into Boost signals.

To use a Boost signal you must connect to a signal and give a method to call when the signal is invoked.

The following examples show how to connect to signals without parameters (preTick), and with parameters (postTick). The binding is done to an object class and an instance of that object.

```
myDe.signal_preTick.connect(boost::bind(&MyObject::method, this));
myDe.signal_postTick.connect(boost::bind(&MyObject::method, this, _1));
```


It is very important to disconnect signals when they are no longer needed or when the object is freed. Failing to disconnect signals can lead to application instability, as, unlike Qt signals, when the object is destroyed the signal is not disconnected.

3.9.1. Application Tick Notification

In VR-Forces 3.12 there was no way to be notified when the application was about to tick or had completed a tick. In VR-Forces 4.x, you can get this information. The main class of a VR-Vantage application is *DtDe*. The *DtDe* class is typically passed to any object that might need to use it, and is available to any plug-in on startup.

To be notified when the application is about to tick, connect to the signal_preTick signal.

On a post tick, connect to the signal_postTick signal.

3.9.2. Selection Management

In VR-Forces 3.12, to find out when selections had changed, you would connect to the Qt signal sent by the map area on selection changed (selectionUpdated, selectionChanged, and so on). In VR-Forces 4.x, you use the Selection Manager and its signals to be notified when selections have changed. You can get a reference to the Selection Manager as follows:

```
DtSelectionManager& selManager = DtSelectionManager::instance(myDe);
```

The signal_currentSelectionChanged signal is sent any time the current selection is changed. It supplies the following parameters:

- The current selection.
- What was unselected.
- The new selection type.
- The old selection type.

The IDs supplied are element IDs, which can be used to retrieve data about the selected objects. "Managing Object Selection," on page 3-3 explains how to use this information.

The signal_vertexSelected and signal_vertexDeselected signals are sent when vertices on a line are selected or unselected

3.9.3. Tracking New Objects

In VR-Forces 3.12, to find out when an object was added, you would have used the modelDataAdded signal. In VR-Forces 4.x you use the *DtElementData* object. You can get to the *DtElementData* as follows:

```
DtElementData& elementData = myDe.dataBank().elementData();
```

The signal signal_elementsAdded is sent for any elements added during that tick.

You can then use the State View Collection Manager to find information about the elements added. Due to the nature of the multi-threading of the application, it is possible that the state view data does not yet exist for those elements. In that case, for the state view collection you want to retrieve information for, connect to the signal_stateViewAdded signal. For example:

```
DtVrfStateViewCollectionManager::instance(myDe).
  baseStateView()->signal_stateViewAdded
```

When the state view is added you can then get information about that object.

3.9.4. Tracking Object Updates

The object data interface allows you to change how often you want to have data refreshed into the object and to be notified when that data is updated. The setUpdateRate() method lets you set the update frequency. The boost signal_dataUpdated signal is sent when data has been updated.

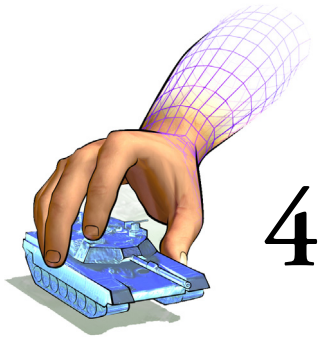
3.9.5. Tracking Resources

To track resources for entities, you can connect to the signal_resourcesChanged() method of the data interface class and then inspect the resources() method to see the resources. You can request new resources by calling the requestResources() method.

3.9.6. Tracking Object Removals

In VR-Forces 3.12 you would have used the modelDataAboutToBeRemoved signal. In VR-Forces 4.x, you use the element data's signal_elementsToBeRemoved signal. This provides a list of all the elements that are being removed in this tick. Please see "Tracking New Objects," on page 3-8 for how to retrieve a reference to the *DtElementData*.

You can also track the removal of specific objects. The boost signal_dataRemoved is sent from the data interface when a simulated object is removed.



4

Simulation Model Set Changes in VR-Forces 4.0.4

This chapter summarizes changes to system definitions and OPE files.

4.1. Simulation Model Set Changes

Changes to the SMS are organized by component or system type.

4.1.1. Flight Command Controllers

Air domain Movement Systems. The flight-command-controller was added to Fixed-wing and rotary-wing entities to handle new flight tasks. (For details, please see fixed-WingFlightCommandController.h and rotaryWingFlightCommandController.h.) If you have created your own air domain movement systems, you must add this controller to take advantage of these new tasks and behavior.

4.1.2. Weapon Interface

The target-priorities parameters have moved. They are no longer part of the target selection controller. They are now in the weapon controllers inside the weapon systems, as follows:

- **target-selection-controller**:
  - Target priorities have been removed from the target selection controller (moved into weapons).
  - The `target-select-controller:weapon-system` connections have been removed. The **target-selection-controller** to sensor connections remain.
  - The controller uses *DtComponentDescriptor* (component-descriptor) for the descriptor type).
- Weapon systems:
  - Target priorities have been added into the weapon controllers. The **target-selection-criteria** parameter has been renamed to **targeting-control**.
  - Target priority metadata has been removed.
  - For ballistic weapons, the **system:target-list** connection is now `<controllerName>:target-to-acquire`. Please see *125mm-gun.sysdef* for an example.
  - Missile Weapons. The `system:target-list` input connection has been removed.

Laser designators have the same changes as weapon systems. They also now have the parameter **integrated-with-launcher**, which determines how target input works (either from the launcher controller, if integrated, or the target selection controller directly if not).

Apache. The laser designator that went along with the laser-guided-hellfire-missile launcher has been moved into the launcher.

The laser-guided-hellfire-missile-launcher system has the following changes:

- Uses the new controller integrated-laser-guided-launcher-controller.
- The laser designator that used to be on the entity is now in this system.
- A new connection (connect missile-launcher:target-to-acquire laser-controller:target-list) was added to represent the missile launcher providing targets to the laser designator.

4.1.3. Radar Modes

Emitter systems now defines a radar-mode-list, each one of which defines a beam-list so these can be switched at runtime. For more information, please see Section 7.5, "Configuring Emitters", in *VR-Forces Configuration Guide*.

4.1.4. Weapon Enhancements

In weapon systems the munition-wrapper resource "ammo" has been removed. Individual munitions are now regular resources.

Ballistic guns have two changes:

- Burst fire. Some ballistic guns now fire in bursts. The rounds-per-minute and extra-time-between-bursts parameters were added to the ballistic gun controller.
- Magazine-based reloading. The rounds-per-magazine parameter was added for magazine-based reloading.

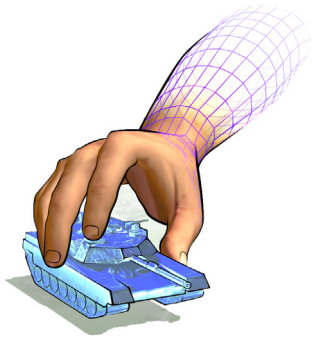
4.1.5. Range Rings

- Weapon systems. The range-name parameter was added to ballistic weapon actuators and missile launcher controllers. This is the display name for range items associated with this weapon displayed in the GUI.
- Aggregate OPE files. The range-name parameter was added. The combat-range-controller was added to define range rings that can represent disaggregation ranges or subordinate weapon systems.
- Munition OPE files. The range-name parameter was added to the missile/torpedo entity definition to display missile range.

4.1.6. Ammo Select Tables

For weapons that fire a burst, hit probability is for the whole burst.

New weapons have been added, so if you have made your own damage system, you may have to make a new damage table to add new mappings.



Index

E

I

K

L

M

N

O

Q

R

S

T

U

V

Link - Simulate - Visualize

68 MOULTON STREET CAMBRIDGE, MA 02138 617.876.8085 www.mak.com