MÄK VR-Forces 3.1-ngc Release Notes

This release note contains the following release-specific information for VR-Forces Release 3.1-ngc:

Document ID: VRF-3.1-3-021016

Systems Supported

VR-Forces 3.1-ngc is available for the platforms listed in Table 1. For the availability of other platforms, contact your MÄK salesperson. For toolkit users, application code must be built with the indicated compilers in order to link to VR-Forces libraries.

Table 1: Platforms supported

Platform	Compiler
SGI IRIX6.5	MipsPro7.3 C++ compiler (available from SGI)
Linux 7.2	GNU C++ (GCC) 3.0.2
PC with Windows NT 4.0 SP6 or Windows 2000 SP2	Microsoft Visual C++ 6.0, Service Pack 5

Disk Space Requirements

A full installation of VR-Forces requires approximately 275 MB of disk space. Terrain databases use approximately 62 MB and documentation (primarily the class reference) uses 102 MB.

MÄK Product Compatibility

To build VR-Forces applications, you must link with VR-Link 3.7.3-ngc. If you are building for HLA and want to link with the MÄK RTI, use MÄK RTI 1.3.7-ngc or later.

MÄK VR-Forces 3.1-ngc is a parallel release to MÄK Plan View Display 2.1-ngc.

Qt Release Compatibility

The VR-Forces GUI was built with Qt release 3.0.5. A VR-Forces developer's license includes a Qt development license. MÄK provides you with a Qt license key. However, you must download the correct version of Qt from www.trolltech.com.

Online Help

The online help is in HTML format and uses the default browser for your computer. For all online help features to work, you must enable Javascript in your browser.

Network Compatibility

HLA only

VR-Forces 3.1-ngc was built against VR-Link 3.7.3-ngc and is compliant with:

- RPR-FOM 1.0 and a subset of 2.0 (draft 6)
- MÄK RTI 1.3.7-ngc
- DMSO RTI NG 4.0, and 6.0.

VR-Forces 3.1-ngc is compatible with applications that use earlier versions of VR-Link if they support versions of the RPR FOM listed above.

DIS only

VR-Forces 3.1-ngc supports DIS 2.0.4, IEEE 1278.1, 2.1.4, and IEEE 1278.1a, and can therefore interoperate with DIS applications of any of these versions.

Backward Compatibility

VR-Forces 3.1-ngc is highly compatible with release 3.0. Programs built with the release 2.x API will need some porting to be compatible with release 3.1. Scenarios created with release 2.x or later are compatible with release 3.1-ngc.

RPR FOM Versions Supported

VR-Forces 3.1-ngc has built-in support for versions 1.0 and 2.0 (draft 6) of the RPR FOM. It also supports VR-Link's ability to support alternative FOMs through the FOM Mapper. By default, VR-Forces 3.1-ngc uses RPR FOM 1.0.

If you are building an application with the VR-Forces toolkit and you want to specify a version of the RPR FOM through code, pass the version number (2.0) to the *DtRprFomMapper* constructor and pass the resulting object to the *DtExerciseConn* constructor. Also, make sure you are using a federation execution name that corresponds to the right FED file. For example:

```
DtExerciseConn conn("VR-Link20", "MyApp", new DtRprFomMapper(2.0));
```

RTI Support

The release is compatible with the MÄK RTI 1.3.7-ngc.

New Features and Updates

This release of MÄK VR-Forces has the following updates and new features:

- Inclusion of the VR-Forces GUI Toolkit header files and examples
- An Echelon View window in the VR-Forces GUI
- Ability to selectively turn paper map layers on and off
- Import of vector data into GDB databases
- Support for subtasking
-

Ability to Selectively Turn Paper Map Layers On and Off

The Terrain View Options dialog box, Drawing Mode tab allows you to selectively display or hide the bitmaps in a paper map. The *./data/terrain* directory includes a new paper map file, *california.map* that implements paper map layers. This feature is fully described in *MÄK VR-Forces User's Guide*.

Import of Vector Data into GDB Databases

The tdbtool has new options for importing vector data into GDB databases. We include the files needed to import DFAD and DFD data into GDB databases. The tdbtool options are described in *MÄK VR-Forces Developer's Guide*. The technical aspects of vector data import and how to access the information is described in *MÄK VR-Forces Developer's Guide*

Support for Subtasking

VR-Forces now supports the subtask flag in *DtSimTask*. This makes it possible to create new types of composite tasks from existing tasks. The task controller for the composite task can task the entity to do each of the subsequent subtasks. Normally, when start-Task() is called, each controller is informed of the new task via a taskNotify() call and clears any task it might be performing. However, if the subtask flag is set, clearTask() is not called. Also, when taskComplete() is called, a request for the next task in a plan may be requested. If the subtask flag is set, this request is not made, and the controller that sent the subtask can instead send the next subtask (or call taskComplete itself, when all subtasks are completed.)

addTaskSim Example

We have updated the userTask example and renamed it addTaskSim. The userTask example showed how to add a new task controller to an entity in the back-end, configurable from the front end through the User Task dialog box. It added a primitive 'retreat task' behavior to an entity in the backend. The addTaskSim example shows how to set up a task controller that responds both to *DtUserTasks* (the generic task class) and to a completely new derived kind of *DtSimTask* from the GUI. It works with the new addTaskGui, described in the GUI API documentation (that example adds a new dialog box for configuring and sending the new retreat task).

Installation Note for Linux and IRIX

This release of VR-Forces includes an updated *vrlink3.7.3-ngc/mkinc* directory. Copy the updated *mkinc* directory into your VR-Link installation. The changes are needed to support building with Qt.

Miscellaneous Changes

- The vrfSimUser project is now in *./appsrc/vrfSim*, instead of *./appsrc/vrfSimUser*.
- To task or set an aggregate entity, you now select the aggregate icon on the map or, if the aggregate is expanded, in the Echelon View. Tasking the aggregate leader does not task the aggregate as a whole.
- When you build VR-Forces, you can ignore errors about undefined environment variables.
- The `-T` command line option, which loads a terrain database, has been extended to apply to vrfGui.

Documentation

All VR-Forces documentation has been updated for this release. This section contains additional updates and error corrections.

Installing the Class Reference

On Windows, the class reference is installed as a self-extracting executable file, *./doc/vrf3.1classref.exe*. If you want to use the class reference, extract the files into the *./doc/classdoc* directory.

Updates to *MÄK VR-Forces Developer's Guide*

Section 6.4 states that callbacks are responsible for deleting messages. This is incorrect. The sender of a message is responsible for deleting it.

The example code in Section 6.5.1 should end with the following line and accompanying note:

```
delete reportMessage;
```

In this example, the radio message is explicitly deleted after the message has been sent. It is the caller's responsibility to delete this radio message.

Viewing Lifeforms in the MÄK Stealth

If you want to view the various postures supported by VR-Forces in the MÄK Stealth, load the *infantryModels.mtl* file by adding it to Stealth.mtl, or using the command line option: `-l config/infantryModels.mtl`.

Configuring Run Duration in Batch Files

When you specify the values for the run-duration parameter in a batch file, hours, day, and minutes must be integers, for example, 15. The seconds value must be real number, for example, 15.0.

Bug Fixes

This release fixes the following problems:

- Late-joining front-ends show the list of back-ends.
- DIS and HLA now use the same default site and application numbers, so you can load example scenarios in either protocol when you are not in combined mode.
- Front-ends that did not create a scenario can manipulate entities and objects in it.
- When you run with multiple front-ends, they all update the state of the buttons in the simulation toolbar.
- Multiple selection of entities using Control-click did not work.
- When you opened new map windows, they did not enable the menus.
- When you closed a scenario, associated windows, such as View Plan, did not always close.
- When you saved a scenario, the *.scn* extension was not added automatically.
- Artillery and missile launchers did not work.
- When you edited a control object, sometimes the wrong type of dialog box opened.
- The front-end did not always drop closed back-ends from its list.
- When you saved a scenario, VR-Forces did not display a confirmation prompt to prevent inadvertent overwriting.
- If you loaded a scenario with a bad terrain database file name, the GUI crashed.
- Icon scaling affected all GUI windows, rather than just the active window.
- The set posture request did not change postures as viewed in the MÄK Stealth.
- Shift-right-click for zooming did not work correctly. The zoom function was not released.
- The Load and Save scenario dialog boxes did not allow you to view all files in a directory.
- The front-end was able to send spurious save scenario messages. The back-end crashed when told to save a scenario that did not exist.
- The Set Heading request used radians instead of degrees.
- The Patrol Between Waypoints task did not let you select the second point in the map. It also let you select the same point for both points.
- The Follow Entity task processed the behind setting incorrectly.
- Task and Set dialog boxes now accept pressing Enter as the equivalent of clicking OK.

- Clicking on the Minimap could hang the GUI.
- The Entity Information dialog box did not always reflect changes to the measurement units setting.
- You could not add the Move Along Route task to a plan.
- When you dragged an entity across the GUI, its height reset to the level of the terrain at its new location. This would cause air entities to crash.
- You could not edit points in a route.
- VR-Forces did not support the alpha channel for *.png* files.
- The Z value and number of rounds were not set for FFE tasks.
- Batch mode works correctly.
- You can load FLT files in succession without crashing VR-Forces.
- Paper map files are supported. However, if the coordinates in the paper map file do not have three data points, the file will not load. See updated documentation of paper map files in *MÄK VR-Forces Developer's Guide*.
- Surface entities follow other entities correctly.

Known Problems

VR-Forces Release 3.1-ngc has the following known problems:

- This release does not support line-of-sight calculations in the GUI.
- This release does not support the DMSO RTI.
- If you are using VR-Forces in a DIS exercise you may experience problems with dropped packets and entities may time out. In HLA, you can avoid dropped packets by configuring the MÄK RTI to use asynchronous IO.
- Aggregates composed of fixed-wing entities do not carry out tasks as an aggregate.
- If you are using a DTED database and "Projected False" is specified in the PvDtedMetafileRecord, entities do not execute movement tasks.
- The *world.gdb* file is based on a DTED database. If you use this database, some entities may not execute tasks. Some entities will not display properly.
- When you aggregate a set of aggregates into a higher-echelon aggregate using the "Aggregate As" menu item, make sure that the aggregates being combined have been collapsed to their highest level. Failure to do so will aggregate the entities at the level they are currently displayed. For example, suppose that you want to aggregate two teams of two tanks each into a platoon. If at the time of the "Aggregate As" operation, one of the teams is expanded, displaying its constituent tanks, while the other team is collapsed, displaying a single team icon, the resulting platoon will be composed of two individual tanks and a team, instead of two teams.
- Surface entities are not configured to fire weapons or be fired upon. You can customize the object parameters database entries to enable this.
- Joysticks do not work correctly with DirectInput.

- The pseudo-aggregate formation controller ignores entities in a formation that are independently tasked when determining if the promotion order needs to be updated. Currently it has no way to determine if remote entities are independently tasked. One possible solution might be to listen to task and task complete messages for all the subordinates in order to maintain a local picture of what is going on.
- Uninstalling VR-Forces in Windows via InstallShield uninstalls the entity fonts. If you have another application that uses these fonts, such as the MÄK Plan View Display, the correct entity icons would not be displayed. You must manually install the fonts from their location in the other application that needs them.
- You cannot control an IRIX back-end from a Windows front-end.
- If you use true type fonts, some entities do not display the correct icons.
- If you try an open a file that is not a VR-Forces scenario file, vrfGui crashes.
- A DtedMetafileRecord that does not have a valid File entry hangs vrfGui.
- If you try to complete a task dialog box by clicking an entity in the map, and the entity icon overlaps a control object, the selection does not take effect. You will need to select the entity in the task dialog box list.
- The Skip Task command does not support multiple entity selection. If you select skip task when multiple entities are selected, only one of them will actually skip their task.
- Sometimes when you load a scenario some of the entities are not displayed. When you start the simulation, the missing entities get displayed.
- The QT translation files (in *./translations*) do not work. The VR-Forces and Plan View Display translation files work correctly.
- The Abandon Plan command does not work.
- If an entity is executing a movement task and you assign any of the Wait tasks to the entity, it does not stop moving. Wait tasks in plans work correctly.

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 *MÄK VR-Forces Developer's Guide*.

Overview of the Main Classes in the GUI API

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

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.

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*.

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:

1. A *DtVrfGuiFactory* is passed into init(), which the *DtVrfGuiApplication* class uses to drive the construction of many of the objects used in the application.
2. The call to the application's fileNewWindow() function creates the main window (a *DtVrfGuiWindow*).
3. 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).

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 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.

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).

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.

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 do to in the terrain map, we need to create a new kind of *DtSymbolFilterLens* (*symbolFilterLens.h*). *DtSymbolFilterLens*' 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 13 for a discussion of how to do this.

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 15). 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 RetreatTask 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 13 for details.

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 13 for details.

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 *DtMtlSymbolMapper* 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 13 for details.