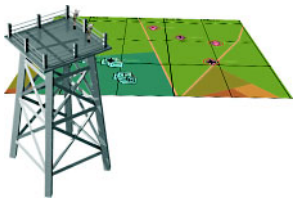


# MÄK Plan View Display 2.1-ngc Release Notes

This release note provides the following release-specific information for MÄK Plan View Display 2.1-ngc:


- Supported Systems and System Requirements ..... 2
  - MÄK Product Compatibility ..... 2
  - Qt Release Compatibility ..... 2
  - System Requirements for PCs ..... 2
  - Online Help ..... 3
- Network Compatibility ..... 3
- RTI Support ..... 3
- New Features and Changes ..... 4
  - Echelon View Window ..... 4
  - Working with Aggregate Entities ..... 5
  - Display of Paper Map Layers ..... 8
  - Loading A Terrain Database at Startup ..... 9
- Bug Fixes ..... 9
- Known Problems ..... 9
- The Plan View Display API ..... 10
  - Overview of the Main Classes in the API ..... 10
  - Customizing or Extending the PVD Application ..... 12
  - Miscellaneous Qt Issues ..... 13
  - Building the pvd Application and Examples ..... 14
  - A Plan View Display API Example ..... 15



Document ID: PVD-2.1-2-021022

# *Supported Systems and System Requirements*

Plan View Display 2.1 is being released for:

- PCs with Windows 2000/XP/NT 4.0
- Linux
- SGI IRIX.

If you are using the Plan View Display API, application code must be built with the indicated compilers in order to link to Plan View Display 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 |

## MÄK Product Compatibility

To use the Plan View Display API, 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 Plan View Display 2.1 is a parallel release to MÄK VR-Forces 3.1-ngc.

## Qt Release Compatibility

The Plan View Display GUI was built with Qt release 3.0.5. A Plan View Display 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.

## System Requirements for PCs

To install the Plan View Display on a PC, you must have a system with the following minimum requirements:

- Processor: Intel Pentium 400 MHz or equivalent.
- RAM: The minimum required for your operating system, plus:
  - 32 MB for DIS exercises
  - 64 MB for HLA federates

Depending on the size of your terrain databases, you may need more RAM.

- Hard disk space - Approximately 120 MB. Most of this is for the terrain databases shipped with the Plan View Display. You may need considerably more space depending on the databases you plan to use.
- Two or more multiples of the amount of RAM for virtual memory. Running the Plan View Display in HLA may require increased amounts of virtual memory.

### 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***

Plan View Display 2.1 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.

Plan View Display 2.1 is compatible with applications that use earlier versions of VR-Link if they support versions of the RPR FOM listed above.

***DIS only***

Plan View Display 2.1 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.

## *RTI Support*

The MÄK RTI is included on all MÄK product CDs. It is link-compatible with the DMSO RTI NG. Plan View Display 2.1 has been tested with DMSO RTI NG 4.0 and 6.0.

If you use the MÄK RTI, be sure that ~~the~~ Plan View Display can find your FED file, and optional *rid.mtl* file, by putting them in the directory from which you are running, or by setting the environment variable RTI_CONFIG to the directory that contains them.

## *FOM Support*

Plan View Display 2.1 has built-in support for versions 0.5, 0.7, 0.8, 1.0, and 2.0 of the RPR FOM. It also supports VR-Link's ability to support alternative FOMs through the FOM Mapper. By default, Plan View Display 2.1 uses RPR FOM 1.0.

The keywords for specifying different versions of the RPR FOM are:

- RPR FOM 0.5: `vlRPR05`
- RPR FOM 0.7: `vlRPR07`
- RPR FOM 0.8: `vlRPR08`
- RPR FOM 1.0: `vlRPR10`
- RPR FOM 2.0: `vlRPR20`.

For information about FOM mapping and selecting the correct FOM mapper, see Startup Options for HLA Federations in Chapter 3, "Getting Started", in *MÄK Plan View Display User's Guide*.

# New Features and Changes

This release has the following new features and changes:

- An Echelon View window, which provides a hierarchical display of the order of battle.
- Display of aggregate entities.
- Display of VR-Forces control objects.
- Ability to selectively display paper map layers.
- Ability to load a terrain database from the command line.
- The Plan View Display API, which enables you to modify the Plan View Display GUI or embed PVD functionality in your own GUI.

## Echelon View Window

The Echelon View window provides a hierarchical view of entities. If an exercise contains aggregate entities, you can expand and contract the entities to display or hide their subordinates.

### Selecting Entities in the Echelon View

The Echelon View window displays entities in a hierarchical view by force. If the order of battle includes aggregate entities, you can expand and contract the display to view or hide subordinate members of the aggregate. The display includes the entity icon and echelon ID.

To select an entity in the Echelon View:

1. Choose Entities → Echelon View. The Echelon View window opens (Figure 2).
2. In the entity list, click the entity you want to select. The Entity List supports multiple selection, as follows:
   - Click to select
   - Shift-click to select a contiguous range of entities
   - Ctrl-click to select multiple non-contiguous entities.

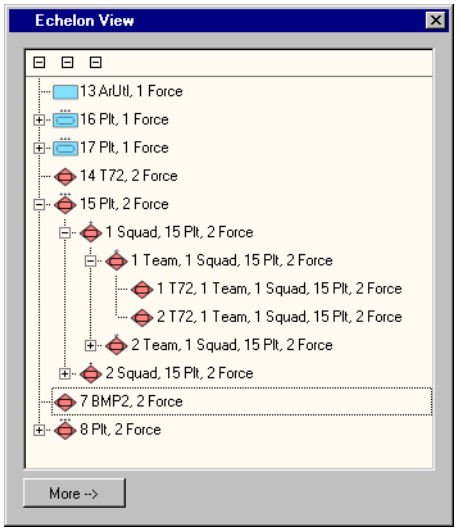



Figure 2. Echelon View window

## Working with Aggregate Entities

The Plan View Display displays individual entities and aggregate entities. Aggregate entities (also called pseudo-aggregate entities) are organizational units created by associating some number of individual entities, aggregate entities, or both. You can display aggregate entities in collapsed format, with just one icon representing the aggregate, and in expanded format, in which the individual members of the aggregate are visible in the map.

### Selecting An Aggregate

➤ To select an aggregate, select its icon on the map, or select it in the Echelon View.

### Expanding and Collapsing Aggregates

You can display aggregates in expanded or collapsed mode. When an aggregate is collapsed, it is displayed as one icon. When an aggregate is expanded, all the members of the aggregate are visible as individual entities.

To expand an aggregate:

1. Select the aggregate you want to expand.
2. Choose Entities → Expand Aggregate.

To collapse an aggregate:

1. Select the aggregate leader.
2. Choose Entities → Collapse Aggregate.

### Expanding and Collapsing Aggregates in the Echelon View

You can expand and collapse aggregates in the Echelon View.

➤ To expand an aggregate in the Echelon View, click the plus sign next to its icon.

➤ To collapse an aggregate in the Echelon View, click the minus sign next to its icon.

### Expanding and Collapsing the Echelon View by Level of Aggregation

In addition to expanding and collapsing the order of battle on a node-by-node basis, as in typical hierarchical view windows, the Echelon View enables you to view entities by the level of aggregation.

Individual entities are directly subordinate to the Force they belong to. If an entity becomes part of an aggregate, the aggregate is subordinate to the force and the individual entity is one level further away. Seen another way, you can say that the force is the parent and all other entities in that force are children, grandchildren, and so on.

The Echelon View window displays the entities in columns that are equivalent to their relationship to the force - the first column is children, the second column is grandchildren, and so on.

At the top of each column in the Echelon View is a plus (+) or minus (-) icon that works like the icons within the hierarchical display. A plus indicates that at least one entity in that column is an aggregate entity and that none of the aggregates in the column are expanded. A minus indicates that at least one aggregate at that level is expanded. Clicking a minus sign collapses all aggregate entities in the column. Clicking a plus sign expands all aggregates in the column.

➤ To change the hierarchical level view in the Echelon View window, click the level controls in the horizontal band across the top of the window.

Figure 3 illustrates an Echelon View window with several levels of aggregates, some of which are collapsed and some of which are expanded.

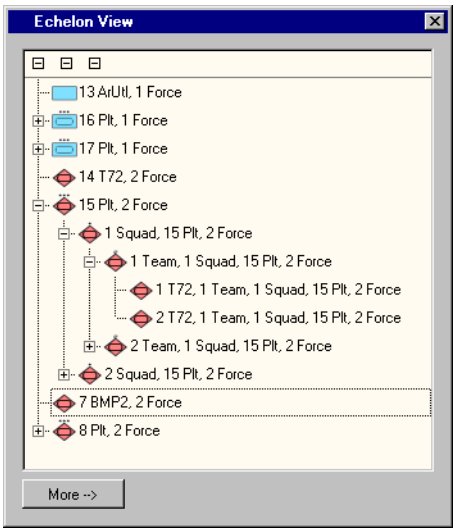



Figure 3. Echelon View window

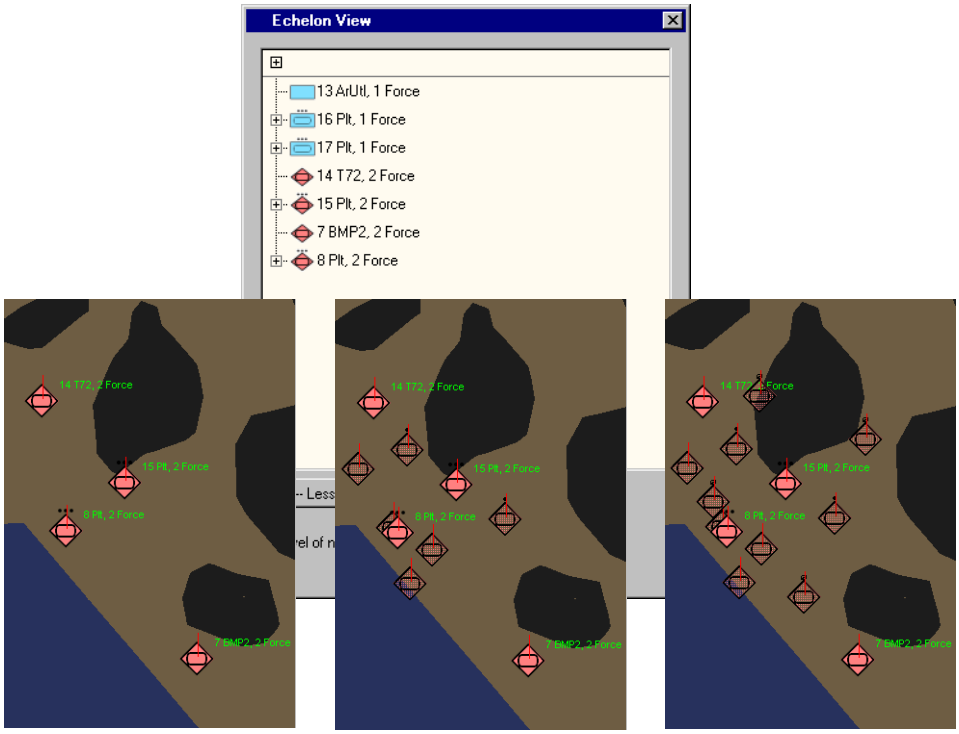
### Displaying Ghosted Icons for Collapsed Aggregates

If you want to select aggregates from the map, you need to collapse them so that you can select the aggregate icon. However, when you do this, you can no longer see where individual subordinate entities are located and if subordinates are destroyed, you cannot see which ones are destroyed and which are functional. You can enjoy most of the benefits of expanded and collapsed aggregates at the same time if you display subordinates as ghosted icons.

When you display ghosted aggregates, the aggregate icon is displayed normally, and the subordinate members are displayed using semi-transparent icons. Figure 4 illustrates the display for a fully collapsed order of battle at three levels of nesting. You cannot select ghosted icons or otherwise interact with them.

To display ghosted icons:

1. Open the Echelon View window.
2. Collapse and expand aggregates as desired.
3. In the Echelon View window, click the More button.
4. In the Level of Nesting Displayed box, select the level of nesting for which you want to view ghost icons. Ghost icons are displayed only for collapsed aggregates.




Level of nesting = 0 Level of nesting = 1 Level of nesting = 2

Figure 4. Ghosted icons

## Display of Paper Map Layers

If you are displaying a paper map, you can specify which layers to display.

To display or hide paper map layers:

1. Choose Options → Terrain View. The Terrain View Options dialog box opens.
2. Click the Layers button. The Paper Map Layers dialog box opens. It lists each of the layers in the paper map file.

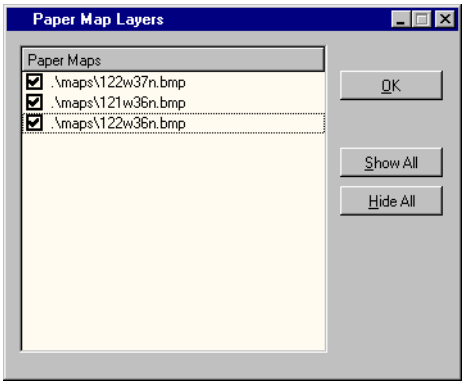



Figure 5. Paper Map Layers dialog box

3. To hide or display a particular layer, check or clear the check box next to the layers name in the list. The display changes immediately.

   To hide all layers, click the Hide All button.

   To Show all layers, click the Show All button.

The *./data/terrain* directory includes a new paper map file, *california.map*, which implements paper map layers.

### Loading A Terrain Database at Startup

You can now load a terrain database at startup using a command line option, `-T`.

```
-T terrain_database
```

## *Bug Fixes*

- The selections in the Entity List dialog box and the map are synchronized.

## *Known Problems*

This section lists problems that we have identified, but which do not interfere with overall use of the Plan View Display:

- This release does not include line-of-sight calculations. We plan to include this feature in a future release.
- Changing the measurement unit does not change the measurement unit for contour lines.
- Using **minMaxLatitude** and **minMaxLongitude** in DTED *.map* files does not work. As a workaround, do not specify these variables in your *.map* files.
- Some of the aggregate entries are not mapped correctly in the *symbolmap-ttf.mtl* file.

# The Plan View Display API

The Plan View Display API enables you to build custom front-ends for viewing exercises.

The Plan View Display 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.

## Overview of the Main Classes in the API

This section describes the key classes in the Plan View Display API.

### The Top-Level Classes

- *DtPvdApplication* (*pvdApplication.h*) is the top-level application class. The application contains the main event loop. It uses a *DtPvdFactory* to create various kinds of top-level objects that it owns, such as the main window, the Terrain Database Manager, and the DIS/RPR Connection Manager. There is only one instance of the *DtPvdApplication* class in the application. It inherits from the Qt class, *QApplication*.
- *DtPvdFactory* (*pvdFactory.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.
- *DtPvdWindow* (*pvdWindow.h*) is the main window in a *DtPvdApplication*. 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*
- *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

- *DtPvdMapArea* (*pvdMapArea.h*) is the primary terrain map display in the main window (*DtPvdWindow*). 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 *DtPvdWindow*. It inherits from the Qt class, *QWidget*.
- *DtModelData* (*modelData.h*) provides the non-graphical representation of the various kinds of simulated objects in Pvd, including entities, aggregates, control objects, and fire and detonation interactions. These objects are maintained by the *DtPvdApplication*.

- *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 *DtPvdMapArea* 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 *DtPvdApplication* 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 *DtPvdApplication* 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 *DtPvdMapArea* has-a *DtSymbolLensArea* area. This enables any number of *DtSymbolFilters* to be added to the terrain map, to affect the appearance of symbols. The *DtPvdMapArea* comes configured with a filter in its lens area for resizing icons (*DtPvdResizeSymbolLens*).

## Customizing or Extending the PVD Application

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

```
#include "pvdFactory.h"
#include "pvdApplication.h"
int main(int argc, char* argv[])
{
DtPvdFactory f;
DtPvdApplication a(argc, argv);
if (!a.init(&f))
{
return -1;
}
a.fileNewWindow();
return a.exec();
}
```

*DtPvdApplication* 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 *DtPvdFactory* is passed into init(), which the *DtPvdApplication* 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 *DtPvdWindow*).
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 PVD 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 *drawMissileImpact* example, we only want to create a derived kind of mtl symbol mapper, so we can configure it to create a new kind of drawable for detonations. To do this, we just have to make a single function call to the factory, telling it to create our new kind of MTL symbol mapper, instead of the default, for example:

```
DtPvdFactory factory;
// Configure the factory so that it creates our new kind of symbol
```

```
// mapper
factory.setMtlSymbolMapperCreatorFcn(MyMtlSymbolMapper::create);
DtPvdApplication app(argc, argv);
if (!app.init(&factory))
{
return -1;
}
```

The MyMtlSymbolMapper::create() function is a static function we defined in our new symbol mapper class that returns a new instance of the class.

```
DtMtlSymbolMapper* MyMtlSymbolMapper::create(DtMtlEnvironment* env)
{
   return new MyMtlSymbolMapper(env);
}
```

By setting a different symbol mapper creator function in the factory, whenever the application requests a new symbol mapper object from the factory, the factory will use our new create function to return our derived type of symbol mapper. You can use this approach to overriding the factory for many of the high-level objects used in the PVD 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 Plan View Display 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 formatted files that Qt compiles to source code using its 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 Plan View Display 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 Plan View Display 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 Plan View Display 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 Plan View Display API uses the Qt signal/slot mechanism for communicating between objects. We define numerous signal and slots in the Plan View Display 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 Plan View Display 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 pvd Application and Examples

We deliver a project and Makefile for building the pvd application in *./appsrc/pvd*. The Plan View Display API examples are included in the *./examples/examples.dsw* workspace (combined with all of the Plan View Display back-end example projects).

## A Plan View Display API Example

The Plan View Display API includes an example *drawMissileImpact* (in the *./examples* directory). The 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 Draw Missile Impact Example

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

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 PVD Application", on page 12 for details.