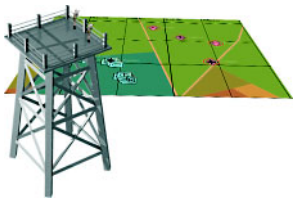

MÄK Plan View Display 2.4-ngc Release Notes

This release note provides the following release-specific information for MÄK Plan View Display 2.4-ngc:

Document ID: PVD-2.4-2-030702

Supported Systems and System Requirements

Plan View Display 2.4-ngc 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)
Red Hat Linux 8.0	GNU C++ (GCC) 3.2
PC with Windows NT 4.0 SP6, Windows 2000 SP2, XP	Microsoft Visual C++ 6.0, Service Pack 5

Building on Linux or IRIX

The example Makefiles require a patched version of gmake 3.80, which has been included for your convenience in the *./mkbin* directory. This version of gmake is based on version 3.80, but includes a memory fix that is needed to use our makefiles. The source code for the modified gmake is freely available from http://ftp.mak.com/out/gmake3.80-patch1.tar.gz. This patch will be included in future versions of gmake.

Before you compile the examples, go to the top level of your installation and create a symbolic link 'VR-Link' to your VR-Link installation and another called 'RTI' to your RTI installation.

MÄK Product Compatibility

To use the Plan View Display API, you must link with VR-Link 3.8.1-ngc. If you are building for HLA and want to link with the MÄK RTI, use MÄK RTI 2.0.1-ngc.

MÄK Plan View Display 2.4-ngc is a parallel release to MÄK VR-Forces 3.4-ngc. It is not compatible with previous versions of the Plan View Display.

Qt Release Compatibility

If you want to do development using the Plan View Display API, you need to use Qt, a cross-platform GUI toolkit. The Plan View Display GUI was built with Qt release 3.1.2. 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.

Online Help

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

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 185 MB. Most of this is for the terrain databases shipped with the Plan View Display and the class reference documentation. 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.

Network Compatibility

HLA only

Plan View Display 2.4-ngc was built against VR-Link 3.8.1-ngc and is compliant with:

- RPR-FOM 1.0 and a subset of 2.0 (draft 6 and 14)
- MÄK RTI 2.0.1-ngc
- DMSO RTI NG 6.

If you neeed to run the Linux version of the Plan View Display with the DMSO RTI, contact MÄK technical support.

Plan View Display 2.4-ngc 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 2.4-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.

FOM Support

Plan View Display 2.4-ngc 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 2.4-ngc uses RPR FOM 1.0.

The keywords for specifying different versions of the RPR FOM with the `-f` startup option 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 darft 6: `vlRPR20006`
- RPR FOM 2.0 draft 14: `vlRPR20014`.

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 Updates

This release of MÄK Plan View Display has the following updates and new features:

- The GUI API has been revised, with the following major changes:
  - New event handler architecture.
  - Elimination of map area classes.
  - Redesigned menu system. Changed how menus are constructed and modified.
  - New examples.

  See "Plan View Display API Migration Instructions", on page 18.
- The Terrain API has been significantly changed, and is now available to developers:
  - The Terrain API now enables developers to create or modify the terrain surface and the vector network.
  - The API has new terrain reader and writer classes and corresponding factories for importing and exporting terrain database files.
  - Plan View Display include Terrain API examples (demonstrating how to create and modify terrain databases). See "Terrain API Examples", on page 7.
  - The *DtSimTerrain* class (a wrapper class around DtTerrainDatabase) has been removed.

  These changes are documented in *MÄK Plan View Display User's Guide*.
- The Plan View Display is now based on standard C++ IO streams.
- Changes and additions to metafile records for vector data.

- New features in the Plan View Display front-end:
  - Support for hiding all control objects.
  - Support for displaying the map in full screen mode, without toolbars.
  - Aggregate Information dialog box.
  - Added Stealth Detach menu item to allow users to explicitly detach the Stealth from a given entity.
  - Echelon View now sorts by designator instead of by name, alphabetically.

Updated GUI API

The updated GUI API is documented in *MÄK Plan View Display User's Guide*. For directions for porting your existing GUI code to the new version, see "Plan View Display API Migration Instructions", on page 18. The GUI API has the following new examples:

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

Terrain API Examples

Plan View Display has three new examples to support the Terrain API:

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

Plan View Display includes the following new terrain databases to support the examples:

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

Changes and Additions to Vector File Metafile Records

The information contained in PvShapeMetafileRecords has been split into two record types: PvShapeMetafileRecord and ShapeVectorMetafileRecord. See "7.4.7 The SHP (Shapefile) Record (replacement for section 7.4.7)", on page 11 and "Update to 7.5.1, Setting Shapefile Parameters", on page 13.

DFAD data is now configured in DfadMetafileRecords. See "DFAD Records", on page 15.

New GUI Features

Plan View Display 2.4-ngc has the following new features in the GUI.

Hiding All Control Objects

You can toggle the display of control objects on and of with the View → View Control Objects menu option.

Displaying the Map in Full Screen Mode, without Toolbars

You can display the map so that it covers the entire screen, with no toolbars displayed. Choose View → Full Screen.

Aggregate Information Dialog Box

If you select an aggregate entity and choose the Entity Information option, it displays information specific to the aggregate, including a list of subordinate entities.

Stealth Detach Menu Option

The Stealth Detach menu item to allows you to explicitly detach the Stealth from an attached entity.

Echelon View Sort Order

The Echelon View window sorts by designator instead of by name, alphabetically.

Documentation Updates

The PDF version of *MÄK Plan View Display User's Guide* is updated to include all new features. Also, note the following clarifications to the printed version of the manual:

MÄK RTI Availability

The printed version of *MÄK Plan View Display User's Guide* states that the MÄK RTI comes with, or is included with, all MÄK products. This statement means that the MÄK RTI software is on product CDs. It does not mean that purchase of the Plan View Display includes a MÄK RTI license. The MÄK RTI is licensed separately from other MÄK products.

Extent Metafile Records

The description of ExtentMetafileRecords is unclear about the role and syntax for the coordinate system value.

CoordinateSystem specifies the coordinate system of the extent you are defining. The SWCorner and NECorner must be expressed in this coordinate system. Recall that Plan View Display is only capable of simulating on UTM databases. So if this extent is going to be used by the Plan View Display simulation engine (as opposed to a Plan View Display GUI that is used solely as an observer), CoordinateSystem must be Utm.

Regardless of the coordinate system used, you must specify a z value for SWCorner and NECorner. Since it is not actually used, you can use 0.0.

Specifying the Coordinate System for Paper Map Records

The coordinate system in which the Paper Map is defined must be aligned with the display coordinate system of the Plan View Display GUI, in order for the features on the map to line up correctly with their locations in the simulated world. (The display coordinate system of the Plan View Display GUI is dictated by the coordinate system indicated in the Terrain Database Record or Extent Record that precedes the Paper Map Record in your *.map* file).

For example, if your underlying terrain database or extent is defined in UTM coordinates, the X and Y axes of your paper map image must be aligned with the X and Y axes of the UTM coordinate system (lines of constant UTM easting and northing should be vertical and horizontal lines respectively, in your map.) On the other hand, if your underlying terrain database is defined in geodetic coordinate, then lines of latitude should run exactly horizontally, and lines of longitude should run exactly vertically in your papermap.

Recall that Plan View Display can simulate only in UTM coordinates, so if you are using a papermap with the Plan View Display GUI to control a Plan View Display simulation, you must use a UTM Terrain Database or Extent, which means that your papermap image must be aligned with the UTM axes.

Specifying the Coordinate System for Pixmap Paper Map Records

CoordinateSystem indicates the coordinate system along which your papermap image is aligned. SWCorner and NECorner must be expressed in this coordinate system. This coordinate system should match the coordinate system of the underlying terrain database or Extent Record. (See "Specifying the Coordinate System for Paper Map Records", on page 9).

SWCorner and NECorner specify the x, y, and z coordinates for the southwest corner and the northeast corner of the papermap. These coordinates must be expressed in the coordinate system designated by the CoordinateSystem attribute described above. In the Geodetic coordinate system, the x and y values define the corners in latitude and longitude (degrees). In the UTM and LCC coordinate systems, the x and y values define the corners in meters. The z value is not used, but you must include it. Setting it to zero satisfies the requirement.

CTDB Metafile Records

The syntax for CTDB metafile records has changed. The syntax is different for the two types of supported CTDB files, as follows:

CTDB Version 4

```
C4bMetafileRecord
{
   File filename
}
```

where File specifies the file name of a CTDB version 4 database. The filename must end with *.c4b*.

CTDB Version 7

```
C7bMetafileRecord
{
   File filename
}
```

where File specifies the file name of a CTDB version 7database. The filename must end with *.c7b*.

7.4.7 The SHP (Shapefile) Record (replacement for section 7.4.7)

To use shapefiles, you define a PvShapeMetafileRecord and, optionally, one or more -ShapeVectorMetafileRecords. For general information about shapefiles, see "Shapefiles," on page 7-16.

PvShapeMetafileRecord

A PvShapeMetafileRecord entry has terrain information and one or more **File** entries. The format for a PvShapeMetafileRecord entry is as follows:

```
PvShapeMetafileRecord
{
  TerrainType {postElevationData | poly | zpoly}
  [SeaRectangleAdded | GroundRectangleAdded]
  Projected {true | false}
  [Origin lat lon]
  [MinimumDistanceBetweenPoints distance]
  [ThresholdForSeaLevel threshold]

  File filename
  elevationFactorToMeters elevation
  {elevationInData | elevationField field_name | elevationValue
  elevation}
  ...
}
```

where:

- **TerrainType** defines how to interpret the data stored in the shapefiles. The possible values are:
  - **postElevationData** specifies that data in the shapefiles is used as points in an irregular grid. The type of shapefile is ignored.
  - **poly** specifies that data is stored as polygons. Valid only with polygon or multipatch shapefiles.
  - **zpoly** specifies that data is stored as polygons with elevation. Valid only with zpolygon or multipatch shapefiles.
- **SeaRectangleAdded** adds a surface at sea level across the entire database. Use this parameter if the shapefile does not include information for the entire area covered by the extent of the database. This parameter is optional and is mutually exclusive with GroundRectangleAdded. See Figure 7-1.
- **GroundRectangleAdded** adds a surface at the lowest ground elevation across the entire database. Use this parameter if the shapefile does not include information for the entire area covered by the extent of the database. This parameter is optional and is mutually exclusive with SeaRectangleAdded. See Figure 7-1.
- **Projected** specifies that the data is UTM (true) or lat, lon (false).
- **Origin** *lat lon* specifies the origin of the database in latitude, longitude, in degrees.

- **MinimumDistanceBetweenPoints** specifies the minimum distance between points, in meters. Used with postElevationData. If two points are closer than this distance, the second one is omitted. Default:150 m.
- **ThresholdForSeaLevel** specifies the elevation below which terrain is considered to be at sea level. Useful when using postElevationData to set new points used to build the GDB and for setting soil types. Default: 0.

The File Entry in A PvShapeMetafileRecord

A PvShapeMetafileRecord must have at least one File entry. The first File entry is used to calculate the origin of the database, if it is not specified with the Origin parameter. If two or more File entries overlap, the Plan View Display uses the highest elevation. The format of the File entry is as follows:

```
File filename
elevationFactorToMeters elevation
{elevationInData | elevationField field_name | elevationValue
elevation}
...
```

where:

- **File** specifies the name of the shapefile. You must have a shapefile (.shp) and dBase file (.dbf) named filename.
- **elevationFactorToMeters** specifies a factor to use to transform the elevation units into meters. Default: 1.
- **elevationInData** | **elevationField** *`field_name`* | **elevationValue** *`elevation`* specifies how to calculate the elevation data, as follows:
  - **elevationInData** specifies that the shapefile contains elevation information, for example, zpoly files.
  - **elevationField** specifies the name of the field in the .dbf file that contains the elevation data.
  - **elevationValue** specifies an arbitrary elevation.

ShapeVectorMetafileRecords

A ShapeVectorMetafileRecord entry specifies feature data information. It has one or more **File** entries. You can specify one or more ShapeVectorMetafileRecords to provide feature data for any of the supported database types, for example, GdbMetafileRecord, DfadMetafileRecord, or PvShapeMetafileRecord.

The origin in a ShapeVectorMetafileRecord must match the origin in the corresponding metafile record.

The format for a ShapeVectorMetafileRecord entry is as follows:

```
ShapeVectorMetafileRecord
{
  Projected { true | false }
  [Origin lat lon]

  File filename
  type type
  ...

}
```

where:

- **Projected** specifies that the data is UTM (true) or lat, lon (false).
- **Origin** *lat lon* specifies the origin of the database in latitude, longitude, in degrees.

The File Entry in A ShapeVectorMetafileRecord

The format of a File entry is as follows:

```
  [File filename
  type type
]
  ...
```

where:

- **File** specifies a file to be imported as feature data. The file must be a .shp file.
- **type** specifies the type of information in the feature data file, according to the description in *./data/importDfad/dfad_sedris_map.txt*.

The color is set based on the file *sedris_colormap.txt*. (See "Importing DFAD or DFD Files into GDB," on page 7-5.) If you want to import feature data with a different specification, add a new entry to dfad_sedris_map.txt.

Update to 7.5.1, Setting Shapefile Parameters

The following is an update to the examples in section 7.5.

Examples

This section has several examples of shapefile records.

```
PvShapeMetafileRecord
{
  TerrainType zpoly
  SeaRectangleAdded
  Projected true
  Origin 47.05 -35.90

  File grb_505_2.shp
  elevationFactorToMeters 1
  elevationInData
}
```

```
PvShapeMetafileRecord
{
  TerrainType postElevationData
  SeaRectangleAdded
  Projected true
  MinimumDistanceBetweenPoints 150
  Origin 17.05 -61.90
  ThresholdForSeaLevel 25

  File ancoast.shp
  elevationFactorToMeters 1
  elevationField ELEVATION

  File antconto.shp
  elevationFactorToMeters 1
  elevationField HTM

}


ShapeVectorMetafileRecord
{
  Projected true
  Origin 17.05 -61.90

  File antspot.shp
  type 774

  File antroad.shp
  type 240

  File acoast.shp
  type 948

  File antcorals.shp
  type 915

  File antponds.shp
  type 940

  File antcliff.shp
  type 924

  File antdrain.shp
  type 945

  File antmangr.shp
  type 953
}


PvShapeMetafileRecord
{
  TerrainType poly
  GroundRectangleAdded
  Projected false

  File gl_eda_elevation.shp
```

```
  elevationFactorToMeters 1
  elevationField ELEV_METER
}
```

DFAD Records

You can import DFAD data directly into the Plan View Display using DfadMetafileRecords. The syntax for the record is:

```
DfadMetafileRecord
{
  [ImportDescriptorFile import_desc_filename]
  [FidSmcFile fid_smc_filename]
  [DfadSedrisMapFile dfad_sedris_map_filename]
  [ColorMapFile color_map_filename]
  [Cropping {true|false}]

  File filename
  [File filename]
  ...
}
```

Where:

- **ImportDescriptorFile** is the name of a file that defines the list of DFAD/DFD features to import. Each possible feature specifies IMPORT or NOIMPORT. By default it loads the file *..\data\importDfad\importFile.txt.*
- **FidSmcFile** is the name of a file that defines the list of FID and SMC codes for DFAD and DFD basic features. By default it loads the file *..\data\import-Dfad\FID_SMC.txt*.
- **DfadSedrisMapFile** - Maps each type of DFAD/DFD feature to a default SEDRIS feature. You can edit this file or specify a different map file with the `tdbTool -e` option. By default it loads the file *..\data\importDfad\dfad_sedris_map.txt*.
- **ColorMapFile** is the name of a file that defines the color representation for each subcategory in the SEDRIS code, in the format RGBA. By default it loads the file *..\data\importDfad\sedris_colormap.txt*.
- **Cropping** is a boolean flag that indicates whether or not to clip the incoming vector data to the extent of the terrain.
- **File** is the name of a DFAD file to load. You can specify any number of files to load.

Example:

```
DfadMetafileRecord
{
  File prague.dfad
}
```

prague.dfad is the name of the DFAD file to load.

DFD Records (Windows Only)

On Windows platforms, Multigen DFD data can be imported using DfdDfadMetafile-Record. This record is almost identical to a DfadMetafileRecord, except that you can specify Multigen DFD files to load in addition to DFAD files. The syntax for the record is:

```
DfdDfadMetafileRecord
{
  [ImportDescriptorFile filename]
  [FidSmcFile filename]
  [DfadSedrisMapFile filename]
  [ColorMapFile filename]
  [Cropping {true|false]

  File filename
  [File filename]
  ...
}
```

Where:

- **File** is the name of either a Multigen DFD file or a DFAD file to import. You can specify any number of files to load.

Example:

```
DfdDfadMetafileRecord
{
  File prague1.dfd
  File prague.dfad
}
```

Bug Fixes

This release fixes the following problems:

- Using **minMaxLatitude** and **minMaxLongitude** in DTED *.map* files works correctly.
- CADRG papermap files display correctly.
- A DtedMetafileRecord that does not have a valid File entry no longer hangs pvd.
- Plan View Display can now load c4b and c7b files from terrain metafiles (*.map* files).
- Fixed a problem drawing contour lines on terrain databases containing reference nodes, which can happen when terrain is imported from OpenFlight files.
- Newly added icons are now always created with the current icon scale, rather than the default icon size.
- The icon scaling slider now works correctly when you click on it to resize, as opposed to click and drag.
- Icon scaling behavior was inconsistent when zooming in or out of the display. Symbols that were not in the view before a zoom operation were appearing with the incorrect scale when brought into view. This has been fixed.

- Plan View Display had problems handling objects with NULL identifiers. Now, if Plan View Display detects an object that has a NULL identifier, it ignores it.
- If you specified too small of a clipping area in DTED metafiles, Plan View Display crashed.
- The aspect ratio of a rubber band was not being maintained correctly when zooming in on an area.
- The log file *pvd.txt* has been renamed to *pvd.log*, to be consistent with other products.
- Zooming out from an aggregate and back in always showed it expanded.
- We removed the 'Entity List' and 'Echelon View' entries from control objects' right-click popup menus. These options were not appropriate in that context.

Known Problems

Plan View Display Release 2.4-ngc has the following known problems:

- On IRIX, icons may appear discolored or grainy unless the color depth is set to 24-bit.
- True Type fonts are not supported in SGI IRIX. Use bitmaps for entity icons.
- If you are using the Plan View Display in a large DIS exercise with heavy network traffic, you may experience problems with dropped packets and entities may time out. You alleviate this problem by using asynchronous I/O by starting with the `-I` parameter.
- Uninstalling the Plan View Display 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.
- The Qt translation files (in *./translations*) for built-in Qt-level GUI items do not work properly. The translation files for Plan View Display work correctly.
- If you are using TrueType fonts, the symbols for air aggregates are incorrect.
- Subsurface entities use incorrect icons.
- Changing the measurement unit does not change the measurement unit for contour lines.

Plan View Display API Migration Instructions

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

New Classes

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

- *DtPvdEventController*
- *DtPvdAppEventController*
- Various menu object (*DtPvdFileMenu*, for example) that encapsulate the creation of menus and menu items
- *DtPvdRightClickMenu*
- *DtPvdMiddleClickMenu*.

These classes are explained in detail in *MÄK Plan View Display User's Guide*.

The Names of Factory Classes have Changed

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

Changes to Existing Classes

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

The *DtMtlSymbolMapper* Class

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

How to Change Code that Subclasses from DtMtlSymbolMapper

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

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

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

The *DtPvdSymbolUpdater* Class

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

How to Modify Code that Subclasses from DtPvdSymbolUpdater

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

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

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

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

Changes to the Menu System

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

The *DtTaskActionGroup* and *DtSetActionGroup* Classes

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

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

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

Old code:

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

New code:

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

or:

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

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

Old code:

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

New code:

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

Replacing the myMapArea Member Variable

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

Migrating Code that Adds Menu Items.

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

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

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

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

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

  return true;
}
```

The code in this example is from a VR-Forces example.

You must change it to look like this:

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

  insertMenuItem(DtNoMenuItem, DtUserMenuItemStart,
    trUtf8( "Retreat..." ), trUtf8( "Retreat..." ),
```

```
      trUtf8( "Task selected entity/aggregate to retreat." ));

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

    return true;
  }
```

Notes

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

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

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

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

  return true;
}
```

Adding Menu Items to Existing Menus

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

- Use the window member function:

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

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

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

The following code is from the fogOfWar example:

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

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

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

  return true;
}
```

- Override the menu.

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

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

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

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

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

  DtPvdDefaultFactory factory;
  DtPvdApplication app(argc, argv);
```

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

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

  app.fileNewWindow();

  return app.exec();
}

#ifndef MYVIEWMENU_H
#define MYVIEWMENU_H

#include "pvdViewMenu.h"

class MyViewMenu : public DtPvdViewMenu
{
  Q_OBJECT

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

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

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

#endif

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

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

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

```
    return false;
  }

  addMenuItemSeparator(0);

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

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

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

Validators for Menu Items

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

(From *popupMenu.h*)

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

```
#include "menuItems.h"

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

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

  // Now add our new item and validation
```

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

  return true;
}

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

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

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

The View Menu

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

1. Subclass the *DtPvdViewMenu* as your own class:

```
#ifndef DtMyViewMenu_H
#define DtMyViewMenu_H

// VRF includes
#include "pvdViewMenu.h"

class DtMyViewMenu : public DtPvdViewMenu
{
  Q_OBJECT

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

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

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

#endif
```

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

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

DtNewMenuItemType(DtMySwitchItem, 1);

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

DtMyViewMenu::~DtMyViewMenu()
{
}

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

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

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

  return true;
}

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

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

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

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

```
#include "pvdDefaultFactory.h"
#include "pvdApplication.h"
```

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

int main(int argc, char* argv[])
{
  DtPvdDefaultFactory f;
  DtPvdApplication a(argc, argv);

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

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

  a.fileNewWindow();

  return a.exec();
}
```

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

Right and Middle Click Menus

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

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

```
    addMenuItem(DtEntityEnableJoystickAction);
    addMenuItem(DtEntityDisableJoystickAction, true);
    addMenuItem(DtStealthAttachAction);
    addMenuItem(DtStealthDetachAction);
    addMenuItem(DtStealthTeleportAction, true);
    addMenuItem(DtEntityDeleteAction);
}
```

Event Controllers and Event Handlers

Most of the functionality that was contained in the map areas, applications, and windows has been moved into event controllers (classes such as *DtPvdEventController and DtPvdAppEventController*) and event handlers (classes such as *DtCreateAreaHandler* and *DtNavigationHandler*). An event controller is a manager of pieces of functionality (event handlers). It installs and runs events based upon actions taken in the interface. You can also install default handlers that are always present and perform some piece of functionality every tick, or when some keyboard or mouse event takes place (for example, the navigation handler or the drag and drop handler). The event handlers are listed in *eventHandlers.h*.

The *DtEntityInfoDialog* Class

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

How to Modify Code That Subclasses from *DtEntityInfoDialog*

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

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