VR-Forces 3.8 Release Notes

This release note contains the following release-specific information for VR-Forces Release 3.8:

Document ID: VRF-3.8-3-050324

Systems Supported

VR-Forces 3.8 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

Operating System	Compiler
Red Hat Enterprise Linux WS 3	GNU C++ (GCC) 3.2.3
Windows 2000 SP2, XP	Microsoft Visual C++ 6.0, Service Pack 5 Microsoft .NET 7.1

If you are building VR-Forces with .NET on Windows 2000, you must install the latest version of the directX SDK (obtainable from Microsoft).

Patching the Microsoft Visual C++ 6.0 Compiler

VR-Forces is transitioning to use of the Standard Template Library, particularly for lists and map objects. Microsoft Visual C++ version 6.0 has a bug with the STL that is fixed with the supplied *xtree-msvc++patch* file. If you use MS Visual C++ version 6.0, you must install a patch to a standard header file. The patch comes from Dinkumware, Ltd (the company that developed the Standard C++ Library header files for Microsoft), but for convenience, we are distributing it with VR-Forces releases.

The patch fixes a bug that would cause applications to crash if they pass std::sets or std::maps from an application to a DLL or vice-versa, something that IEEE 1516 federates must do. For details about the patch go to http://www.dinkumware.com/vc_fixes.html.

To install the patch:

1. In the top level VR-Forces directory, is a file called *XTREE-msvc++patch*. Rename this file *XTREE*.
2. Make a back-up version of the original *XTREE*.
3. In the Visual Studio *include* directory (for example, *C:\Program Files\Microsoft Visual Studio\VC98\Include*), replace the original version of *XTREE* with the patched version.

Building on Linux

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.

Disk Space Requirements

A full installation of VR-Forces requires approximately 540 MB of disk space. Terrain databases use approximately 106 MB and documentation (primarily the class reference) uses 200 MB.

Compiler Compatibility on Windows

MÄK provides versions of product releases that have been compiled with Microsoft Visual C++ 6.0 and 7.1. When you run MÄK products together, for example, the VR-Forces and the Stealth, we strongly recommend that you run versions compiled with the same compiler. Mixing products compiled with different versions of the compiler can result in program instability.

MÄK Product Compatibility

To build VR-Forces applications, you must link with VR-Link 3.9.3. If you are building for HLA and want to link with the MÄK RTI, use MÄK RTI 2.x.

MÄK VR-Forces 3.8 is a parallel release to MÄK Plan View Display 2.8. It is not compatible with previous releases of the Plan View Display.

Qt Release Compatibility

If you want to do development using the VR-Forces GUI API, you need to use Qt, a cross-platform GUI toolkit from Trolltech. The VR-Forces GUI was built with Qt release 3.3.1. 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*.

Third-Party Library Requirements

DtTerrainProfileWidget is based, in part, on the work of the Qwt project (*http://qwt.sf.net*). To compile examples or user projects, you must download the QWT distribution listed on their home page. Set the MAK_QWTDIR environment variable to point to the location of the QWT library.

Network Compatibility

HLA only

VR-Forces 3.8 was built against VR-Link 3.9.3 and is compliant with:

- RPR-FOM 1.0 and a subset of 2.0 (draft 6 and 17)
- MÄK RTI 2.3.3
- DMSO RTI NG 6.

If you need to run the Linux version of VR-Forces with the DMSO RTI, contact MÄK technical support.

VR-Forces 3.8 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 3.8 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.

Backwards Compatibility

While we attempt to preserve backwards compatibility of the API and configuration between successive versions of VR-Forces to the greatest degree possible, we do have to break backwards compatibility in some situations. This is not a choice we make lightly. In some cases, breaking compatibility with previous releases is necessary to implement a new capability. In other cases, we have chosen to break backwards compatibility because we feel the alternative is vastly easier to use and maintain in the long run.

This section lists the subset of changes that are most likely to potentially affect customer developed code, configuration, or both. For specific details regarding a particular change, see the corresponding section in the release notes.

Changes to the API:

- New road and path planning model classes. See "Path Planning," on page 16.
- Bounding volume class refactored. See "DtBoundingVolume Migration Guide," on page 32.
- Obstruction sensor and collision detector classes were refactored. See "DtCollisionDetector," on page 54 and "DtObstructionSensor," on page 55.
- New aggregate model classes largely replace previous model. See "New Classes for Managing Aggregates," on page 47.
- *DtSpatialSubdivision* class replaces *DtSpatialIndex* class. See "Spatial Subdivisions," on page 41.
- Change in relationship between *DtCgf* and *DtVrfCreator* classes. See "Changes to the DtCgf Class," on page 38.
- Accessor for process state repository in components renamed. See "Object Parameters Database Changes," on page 29.
- Changes to *DtReaderWriter* class interface.
- Task Manager and task controller changes.

Object parameters database:

- Changes affecting potentially all *.ope* files:
  - New non-zero bounding volume requirement affects potentially all parameter database entries. See "Object Parameters Database Changes," on page 29.
  - Relative paths specified in parameter database files are now relative to the *.ope* file, not the application path. See "Changes to Pathname Interpretation in Object Parameters Database Files," on page 12.
- Changes affecting individual platforms:
  - New path planning component descriptors.
  - Changes to damage component descriptor.
  - Components for new centralized fire and detonation processing. See "Centralized Fire and Detonation Processing," on page 31.

- Changes affecting lifeforms:
  - New lifeform acquire descriptor.
- Changes affecting aggregates:
  - New aggregate component descriptors.
  - Aggregate configured with local task manager instead of now obsolete pseudo-aggregate task manager. See "New Classes for Managing Aggregates," on page 47.

Terrain file format:

- Terrain file format version incremented. See "New GDB File Version," on page 51.

Model behavior changes:

- Road and path planning. See "Path Planning," on page 16.
- Aggregates. See "Improved Aggregate Behaviors," on page 18.
- Fixed-wing circle behavior.
- Movement along routes in the event route changes.

Build configuration changes:

- New VR-Forces GUI libraries:
  - *pvdController{5, HLA13, HLA1516}.lib*
- New VR-Link libraries that must be linked in to PVD and VR-Forces GUI applications:
  - *loggerControl{5, HLA13RT, HLA1516RT}.lib*
  - *stealthControl{5, HLA13RT, HLA1516RT}.lib*

The FED file has a new entry to support remote control of the GUI.

RPR FOM Versions Supported

VR-Forces 3.8 has built-in support for versions 1.0 and 2.0 (draft 6 and 17) of the RPR FOM. It also supports VR-Link's ability to support alternative FOMs through the FOM Mapper. By default, VR-Forces 3.8 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 (for example, 2.0006) 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-Link20006", "MyApp", new DtRprFomMapper(2.0006));
```

In order to support RPR FOM 1.0, we have added the following extensions (which are supported in later versions of the RPR FOM):

- EnvironmentProcess
- GriddedData
- EmbeddedSystem.IFF
- EmbeddedSystem.IFF.NatoIFF
- EmbeddedSystem.IFF.NatoIFF.NatoIFFTransponder
- EmbeddedSystem.IFF.NatoIFF.NatoIFFInterrogator
- EmbeddedSystem.IFF.SovietIFF
- EmbeddedSystem.IFF.SovietIFF.SovietIFFTransponder
- EmbeddedSystem.IFF.SovietIFF.SovietIFFInterrogator
- EmbeddedSystem.IFF.RRB
- BaseEntity.AggregateEntity
- ObjectRoot.BaseEntity.PhysicalEntity.Lifeform.

For both RPR FOM 1.0 and 2.0 VR-Forces 3.8 relies on the LgrControl and ViewControl custom MÄK extensions.

New Features and Updates

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

- New GUI features
  - Line-of-sight filtering
  - Map view mirroring
  - Freehand lines
  - Adjustable line width
  - Ability to set entity capabilities
  - Ability to set appearance attributes
  - Dynamic loading of the Create menu
  - Cultural features added to the Create menu
  - Display of transmitter and signal PDUs
  - Range rings
  - Object transparency
  - Displaying entities as dots instead of using icons
  - Papermap transparency
  - Stealth Navigation toolbar
  - Environmental List and Environmental Information dialog boxes.
  - Fire and detonate lines can be configured for display independently
  - Fire and detonate interactions can be hidden
  - Display of hazard clouds
  - Shadow text
- GUI object parameters database editor
- Improved entity modeling
  - Improved aggregate behaviors
  - Improved road following and path planning
- Configuration and startup changes
  - Configuring search paths for scenario files
  - New scenario file parameters
  - Editable default scenario file
  - Configuration file changes and new MTL parameters
  - New startup options
  - New temporary directories for VR-Forces data
  - Ability to set host address
  - Ability to change or hide the background font character

- Terrain database changes
  - Tdbtool updates
  - Support for Flat Earth
  - Ability to set RPF resolution
  - Updated Image Splitter utility
- GUI API changes
  - Remote control
  - Classes for transmitter display
  - Classes for transparency.
  - Symbol mapper updates
  - Appearance attributes
  - Removed classes
  - Changes to FED file for HLA to support view controls
- Component Code Generator utility
- Improved performance for large scenarios
- API changes
  - Improved aggregate movement and formation behavior
  - Changes to Task Manager and task controller
  - Changes to the object parameters database
  - Changes to processing fire and detonation interactions
  - Ability to extend state repositories at the base class level
  - Renamed functions
  - GeometryNative classes removed
  - *DtReaderWriter* class interface changed
  - New examples.

All new GUI and configuration features are fully documented in *VR-Forces User's Guide* and *VR-Forces Configuration Guide*.

Object Parameters Database Editor

VR-Forces has a new utility, the OPD Editor, a GUI-based editor for the object parameters database. The OPD Editor is in *./bin*. It lets you load your object parameters database and edit parameters for individual objects or for multiple objects within the hierarchy. You can change existing parameter values and add and remove components, resources, and soil types. For a complete explanation of how to use the OPD Editor, see "Using the OPD Editor," on page 5-22 of *VR-Forces Configuration Guide*.

Configuration and Startup Changes

This section describes new startup options and changes to configuration files.

Changes to Pathname Interpretation in Object Parameters Database Files

Object parameters database entry files (**.ope*) often contain references to supporting files (such as damage files (*.dmg*) or hit probability files (*.hit*)) that must be loaded. The convention for specifying paths to these supporting files has changed. In previous versions of VR-Forces, the paths to these files were always specified relative to the application path (for example, *./bin5/vrfsim.exe*). Supporting files are now specified relative to the containing file.

The convention was changed to make object parameters database files more self-contained, eliminating the dependency on the application path. You can now move an object parameters database (the top-level *.opd* file and related parameter files) without having to adjust any of the filenames contained in the *.ope* files. Customers have requested this change so that they can maintain custom object parameters databases outside of the VR-Forces file hierarchy and not have to move them every time they install a new version of VR-Forces.

As a result of these changes, the version number for object parameters database files has advanced to version 101. Legacy format files can still be loaded by VR-Forces. However, we recommend that you convert customized object parameters databases to the latest version to ensure compatibility with future versions. Run the *convertParamDb* application to update your legacy format object parameters database (version 100 or pre-3.7.*opd* files) to the latest version.

Configuring Search Paths in Scenario Files

Two new variables have been added to the *vrfSim.mtl* configuration file. The new parameters are:

- **VRF_defaultParameterDatabasePath** – specifies the path to search for *.opd* files when an explicit path is not specified as part of a filename in a scenario file.
- **VRF_defaultTerrainDatabasePath** – specifies the path to search for terrain database files when an explicit path is not specified as part of a filename in a scenario file.

The **VRF_defaultParameterDatabasePath** is used in the search path when loading an object parameters database in the back-end. The object parameters database file is specified in a scenario file. The order in which directories are searched for a given object parameters database are as follows:

1. If the path is absolute, load the file in the given path.
2. If the path is relative, try to load the file relative to the application path.
3. Look in **VRF_defaultParameterDatabasePath**.

The **VRF_defaultTerrainDatabasePath** is used in the search path when loading a terrain database in the back-end. The terrain database file is specified in a scenario file. The order in which directories are searched for a given parameter database are as follows:

1. If the path is absolute, load the file in the given path.
2. If the path is relative, try to load the file relative to the application path.
3. Look in **VRF_defaultTerrainDatabasePath**.
4. If defined, look in the user-defined environmental variable VRF_TERRAIN_PATH.

New Startup Options

- `-F ttl` – specifies the mcastTtl value of the exercise connection socket. (DIS only)
- `-H ethernet_address` – specifies the ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose. (front-end only)
- The `-R` startup option lets you load a script file at startup.

  `-R script_file`

  VR-Forces includes the following MTL commands that the GUI can respond to:

  – (`opendatabase database_name`) – Opens and displays the specified database as if the user selected File/Open Database and chose a database to open.

  – (`loadoverlay overlay_file`) – Loads an overlay file as if the user selected File/Load Overlays.

  – (`setlocalextent` (`list x y z`) (`list x y z`)) – Sets the front end extent (in local coordinates) to the specified North, East, and Southwest coordinates (as if you zoomed into a particular area.)

These commands are only supported locally and are not sent out over the network.

Default Scenario

VR-Forces has a new, user-editable default scenario – *./data/scenarios/default.scn*. This scenario is loaded whenever you start a new scenario. If you want to customize the settings for all new scenarios, edit *default.scn*.

Configuration File Changes

VR-Forces has some new configuration files and has made some changes to existing files, as follows:

- *vrfGui.mtl* now loads *pvd.mtl* as part of the VR-Forces startup. This is so variables do not have to be duplicated in the two MTL files.
- The character column in *symbolmap-uttf.mtl* file that appears after the font name has been removed.
- *capabilities.mtl* – stores settings for entity capabilities.
- *appearance.mtl* – stores settings for entity appearance attributes.
- *rangeRings.mtl* – configures threat rings for entities.

New Configuration Parameters

VR-Forces has the following new parameters:

- **VRF_forceParameterDbReload** – in *vrfSim.mtl*, forces *DtCgf* to reload the object parameters database each time a scenario is loaded. This setting defaults to false, which means that if a scenario is started that uses the same object parameters database as the previous one, the object parameters database is not reloaded. When enabled, this setting makes it easier to develop object parameters databases because you can read in changes you have made to the database without having to restart VR-Forces.
- **PVD_viewControlMessages** – in *pvd.mtl,* enables or disables view control messages. See "Reflection of the Map View," on page 20.
- **PVD_storeLineValuesMode** – in *pvd.mtl*, specifies whether or not to save the settings for newly created lines to apply to the next new line. See "Storing Line Settings," on page 20.
- **PVD_defaultFreehandSamplingRate** – in *pvd.mtl*, specifies the default number of points to sample when drawing a freehand mode line. All points are recorded until the object is created, at which point only the sampled amount of points are saved.
- **PVD_maximumSignalTimeDifference** – in *pvd.mtl*, specifies the number of seconds between when a signal is sent on a certain transmitter frequency and another is sent on the same frequency to try and determine which entities might be responding to an originating signal.
- **PVD_hideOffScreenSymbols** – in *pvd.mtl*, specifies whether or not to hide symbols when they go off screen instead of destroying them. Setting this parameter to 1 may help with performance for large exercises.
- **VRF_hostAddrString** – in *vrfGui.mtl*, specifies an ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose.
- **VRF_defaultParameterDatabasePath** – in *vrfSim.mtl*, specifies the path to search for *.opd* files when an explicit path is not specified as part of a filename in a scenario file.

- **VRF_defaultTerrainDatabasePath** – in *vrfSim.mtl*, specifies the path to search for terrain database files when an explicit path is not specified as part of a filename in a scenario file.
- **VRF_logFrameRateStatistics** – in *vrfSim.mtl*, enables (1) or disables (0) logging of frame rate statistics by the back-end. Output is logged to a file named *vrfSimFrameRateStatistics.site number_appnumber_scenario name.timestring.log*. For example, the following file was generated when running the Berkeley scenario with logging enabled: *vrfSimFrameRateStatistics.1_3001_berkeleydemo.1108945497.log*

Host Address String

You can specify the ethernet address to receive files from the server during a save. This is for use on computers that have multiple network cards and on which you do not want to use the first network card for this purpose. To specify the ethernet address, use **VRF_hostAddrString** in *vrfGui.mtl* or the `-H` startup option with vrfGui.

Temporary Directories

To prevent conflicts in the temporary directories when running multiple copies of VR-Forces from the same directory, VR-Forces now creates unique temporary directories. The directories are created in the path specified by the TMP environment variable. If this variable is not specified on Linux, the */tmp* directory is used. The TMP variable must be specified on Windows. If VR-Forces cannot create a temporary directory, an error message is printed and the application aborts. In the front-end, the temporary directory is created by *DtVrfRemoteController* and starts with the prefix *vrffe*. In the back-end, the temporary directory is created by *DtCgfDispatcher* and has the prefix *vrfbe*. In both cases, the temporary directory is removed when VR-Forces shuts down.

Changing or Hiding the Background Character

You can now change or hide the character used to create the background color of symbols based on TrueType fonts. When you use TrueType fonts for entity symbols or any other symbols configured for the GUI, the background color, for example, blue or salmon, is determined by a character specified in the configuration file. By default, this is the @ character. (If you have ever run VR-Forces without installing the TrueType fonts, you may have seen @ characters displayed instead of the expected entity icons.)

The new configuration option lets you specify a different background character or eliminate it altogether. To specify an alternative background character, add it, in double quotes, to the end of the symbol's entry in the *symbolmap-ttf.mtl* file (or any other configuration file that specifies a "ttf" character. To specify no background, add empty double quotes, as follows:

```
(symbol-map (list  0 -1  -1  3 -1 -1 -1 -1 -1) "ttf" "MapSym NK Sea" "0"
  "")   ;;   Clears out background character
```

This configuration option is optional. You do not have to update existing symbol map files if you do not need to use it.

Entity Modeling

This section describes improvements to VR-Forces' modeling.

Path Planning

The new Path Movement component (*DtPathMovementController*) allows ground entities (vehicles and lifeforms, including aggregate entities) to plan routes across terrain when instructed to move to a destination. This component uses a standard A* search algorithm to find the best route for the entity from among a set of evenly spaced routes along a grid, combined with any road vector features that exist in the terrain. The spacing and size of the grid routes, as well as the types of vector features that are considered in the search is customizable though the descriptor for the component in the object parameters database file. Vector feature types that are to be treated as impassable can also be specified in the object parameters database. By default, most ground entities treat all features in the MAK Roads group as roads, and all features in the MAK Water group as impassable.

The evaluation of each segment of the path is done though the *DtPathMetric* class. The type of path metric to use is specified in the descriptor for the component. There is a standard factory for path metrics, and a user can define their own new path metric classes to change the way routes are evaluated for any particular entity type. VR-Forces ships with a standard Slope and Soil Type path metric that is used for all ground vehicles. It considers the soil type along each path section, as well as the average slope of the terrain, and compares these values to the capabilities of the vehicle to determine the cost of transversing the path.

To task an entity to calculate a path before moving, select the Plan Path movement option in the Move To or Move To Location dialog box. Entities must have the Path Movement Controller to respond to this task.

Depending on the complexity of the terrain, the size and spacing of the search grid, the length of the path, and the available processor power of the computer being used, it may take anywhere from under a second to multiple minutes for an entity to calculate its path. The path planning is done in a separate thread from the main simulation, so it will not affect the running of the simulation. While an entity is calculating its path, it does not move.

The best path that is found from the search algorithm is published in the simulation, and displayed on the map. You can prevent routes from being published, in which case they are not displayed. The entity still follows the route.

You can edit the route and the entity will move along the modified route. Once the route is published, the entity moves along it as it would along any other route. When the entity reaches its destination, the route is removed.

In the object parameters database that ship with VR-Forces, all ground vehicles have a path movement controller. This component replaces the *DtGroundMoveToPathController* and *DtGroundPatrolBetweenPathController* components, which have been removed.

Notes

- Because multithreading is used in this component, any VR-Forces scenario that makes use of path planning will not be able to be run in repeatable mode.
- In previous versions of VR-Forces, civilian vehicles were configured to follow paths when moving to a location. In VR-Forces 3.8, when you give a ground vehicle a move to task, you can specify direct movement or path planning. As a result of this change, if you run a scenario created with a previous version of VR-Forces and that scenario has civilian vehicles with plans containing move to tasks, the civilian vehicles will default to direct movement, rather than path following.
- Use of path planning may degrade performance. Performance will be improved if you use spatial subdivisions (see "Spatial Subdivisions," on page 41).

Road Driving

All entities using the *DtMoveToDestinationControllerComponent* class or its derivatives (all ground vehicles, and dismounted infantry) now have a Stay in Lane mode. You can enable road following in the object parameters database entry for the entity, or though a new Set Stay in Lane set data request.

If Stay in Lane is enabled for an entity, it follows any vector features in the terrain that belong to the MAK road feature group that it encounters as it moves. This means that if the entity's direction of movement is approximately along a road, the entity alters course to drive along the road, offset from the road center by a distance specified in the entity's object parameters database entry. If the road turns, or the direction of movement is no longer along the road, the entity reverts to its usual movement directly toward the destination.

Road driving is different from path planning in that it simply makes vehicles follow the rules of the road when they drive on or near a road. It does not take them out of their way to move along roads. It can be used in conjunction with path planning to keep a vehicle on the right side of the road when it has planned a path using the roads.

The default object parameters database files that ship with VR-Forces disable road driving for all entities except the civilian vehicle to preserve behavior compatibility with previous versions.

Improved Aggregate Behaviors

Aggregate components have been redesigned for version 3.8. The changes to aggregate behavior include:

- Improved move-along-route behavior. Aggregates no longer cut-corners when following a route.
- New Move into Formation task. This tasks an aggregate to move into a particular formation, at a given location and heading.
- Improved set formation behavior. Members of an aggregate no longer try to jockey into position when an aggregate is dragged on the map, issued a Set Location request, or Set Heading request.

Hiding Aggregate Formation Routes

You can hide the temporary routes generated by an aggregate when it is given a route-following task using an option on the View menu.

Terrain Database Changes

The tdbtool has the following new options:

- Databases can now be converted to flat earth using the `-t` option.
- The origin of the database can now be manipulated by using `-setorigin` or `-moveorigin` and specifying the new origin with `-lat <D:M:S>`, `-lon <D:M:S>`, and `-zone <1 - 60>`.
  - `-setorigin` only moves the origin of the database. It does not reproject the database to the new origin (so, basically, moves the local origin.)
  - `-moveorigin` moves the database to the new origin location, changing the coordinates within the database (this is useful for moving a database from one zone to another.)
- New options have been added for specifying the expected number of vertices, surfaces, specifications expected in an imported terrain (any format). This can result in significant memory and performance savings for very large terrain files. The options are:
  - `-numvertices` *`expected_number_of_vertices`*
  - `-numsurfaces` *`expected_number_of_surfaces`*
  - `-numspecs` *`expected_number_of_specifications`*

  Example:

```
tdbtool –numvertices 10000000 –o .\example.gdb
  .\example.flt
```

- The Makland terrain database used in the maklanddemo scenario now uses a database that does not have polygonal features (buildings, trees, light poles). The result is that intervisibility will not recognize features unless you turn on feature intervisibility in the Intervisibility Options dialog box. Furthermore, you must increase the range of interest for intervisibility to recognize buildings. When computing line-of-sight for entity fire, the back-end may not adequately recognize features.
- When you import a terrain database, some default processing is done to remove duplicate vertices and, for non-GDB database, to add a spatial subdivision, if one is not specified.

Flat Earth Support

Flat earth flight files can now be loaded up in the front-end and simulated in the back-end. Previous versions of VR-Forces automatically converted flat earth databases to UTM. They are no longer converted, but are supported as flat earth.

RPF Resolution Keyword Addition

Chapter 2, *Working With Terrain Databases*, in *VR-Forces Configuration Guide* has added documentation for RPF paper map records. These record types have a new keyword that lets you specify the resolution to use. The default values are "1:1M" for CADRG and "5M" for CIB. The syntax is:

```
Resolution "resolution"
```

Component Code Generator

The Component Code Generator (*./bin/componentCodeGenerator.exe*) is a utility that assists programmers in developing new *DtSimComponent* classes for VR-Forces entities. It automates much of the work needed to set up a simulation component, and writes out C++ source files which can then be tailored by the developer for their specific needs. For details, see Chapter 5, *Creating New Components*, in *VR-Forces Developer's Guide*.

Improved Performance

The performance of VR-Forces has been greatly improved when simulating with large numbers of entities, for example a simulation of 1500 ground entities on a relatively complex terrain. To achieve this improvement, spatial organization of both *DtVrfObjects* and the vector feature obstructions has been added. All users should see improved performance of their scenarios, and large simulations will exhibit the most improvement. See "Spatial Subdivisions," on page 41, for more information.

New GUI Features

This section describes new features in the GUI display.

LOS Filter

The line-of-sight filter allows you to view entity line-of-sight within the boundaries of the filter. You can filter by force type and color the lines by force type.

Reflection of the Map View

You can control the map view of other instances of the MÄK Plan View Display and VR-Forces with the view control feature. You can set any individual instance of these applications to ignore view control messages. You can also enable or disable view control messages with the **PVD_viewControlMessages** MTL variable in *pvd.mtl*.

Freehand drawing

You can now draw freehand lines on overlays. Once you draw a freehand line, it behaves like all other lines (for example, an entity can be tasked to move along it).

When you create a freehand line, you can specify the number of points to create along the line.

Once a line has been down-sampled, the points removed during the down-sampling cannot be recovered.

Storing Line Settings

A new configuration parameter, **PVD_storeLineValuesMode**, lets you specify that new lines get created with the same color, width, sampling, and force settings as the previous line that was created. This variable has the following options:

- 0 - Do not store any values
- 1 - Store for freehand lines only
- 2 - Store for all lines.

This variable only applies to line-based overlay objects and is only valid for the current session.

Specifying Line Width

You can now set the width of individual lines. If no width is specified, the default width is used for the particular object type.

For network representation, the high four bits of the appearance of the object is used to represent this width.

Entity Capabilities

You can set entity capabilities (ammunition supply, fuel supply, recovery, repair) with the Set Capabilities set data request. Default entity capabilities are specified in the *capabilities.mtl* file. Developers can set capabilities using the remote control API.

Setting Appearance Attributes

You can set entity appearance attributes with the Set Appearance set data request. Appearance attributes are stored in the *appearance.mtl* file, and are defined from the DIS standard. The *appearance.mtl* file is responsible for displaying attributes both in the Entity Information dialog box and the Set Appearance dialog box.

The Set Appearance dialog box displays the current values. All changes made to the appearance are immediate and are set in the entity state repository without modification or verification.

See "Setting Appearance Attributes," on page 55 for API changes that support this feature.

Loading the Create Menu Dynamically

You can now load alternative Create menu files (*vrf.ent*) during runtime. This does not affect the overlay portion of the menu, only the portion that creates entities and control objects. You can also specify a create menu file in a scenario file.

Ability to Create Cultural Features

The Create menu includes a new sub menu for cultural features. The menu lets you place neutral force objects that represent objects such as bridges, houses, towers, and so on.

Transmitter and Signal Support

VR-Forces now supports display of reflected transmitters (using the VR-Link *DtReflectedRadioTransmitterList*) and captures signal PDUs/interactions to associate signals with the transmitters that might be sending them. The Display Options dialog box has options to display transmitters, display communication lines, and allow for transmitter selection.

You can view information about transmitters by looking at the Transmitters tab on the Entity Information dialog box. The Transmitter Information dialog box shows detailed information about a specific transmitter and any signals that might have been sent by that transmitter.

The transmitter settings are stored in the *settings.mtl* file, so, if you migrate to a new version of the software, you can copy the *settings.mtl* file to the new version to keep those frequency settings.

A new configuration parameter, **PVD_maximumSignalTimeDifference**, allows you to configure the number of seconds between when a signal is sent on a certain transmitter frequency and another is sent on the same frequency to try and determine which entities might be responding to an originating signal.

Threat Circles (Range Rings)

Threat circles show the potential threat radius of an entity. The threat radius is defined in *rangeRings.mtl* and lets you define multiple threat (range) rings for the weapons an entity might have associated with it. The default is to show a ring with cross hairs to more easily show which entity is associated with which range ring.

Range rings have the following display options:

- Drawing of transparent circles (this mode will significantly degrade the performance of the GUI.)
- Displaying of multiple rings leading to the outer ring (in the case of a very large range area, this can help to tie a ring to an entity)
- Displaying of range (distance) labels for the outer and inner range rings.

On small databases, an entity's range may be beyond the edge of the database display, in which case, you will not see it unless you zoom out.

Object Transparency

All overlay objects can now have transparency associated with them. You apply transparency using a slider on the More tab of the Create/Edit object dialog box. You can set transparency from 0 (opaque) to 100 (fully transparent). Transparency does not affect points and highlighting.

Replacing Entity Icons with Generic Symbols

You can replace entity icons force colored dots. This may increase performance. You can also show/hide entity symbols on a per-entity basis or for a selected group of entities.

Papermap Transparency

A new slider has been added to the Raster Map layers dialog box to make setting of transparency levels easier. Select the layer and move the slider to change the transparency level. If the Apply Changes Immediately check box is checked, any change on this dialog box take effect immediately.

Stealth Navigation Toolbar

The new Stealth Navigation toolbar lets you control the MÄK Stealth eyepoint and attachment modes, if there is a broadcasting Stealth available. The toolbar sets the mode of the selected Stealth or, if there is only one Stealth broadcasting, that Stealth.

If the Stealth Information dialog box has focus, you can use the keypad to move the Stealth in the same way as by using the keypad in the Stealth.

The Stealth Navigation toolbar requires the use of the MÄK Stealth 5.3 or higher.

Displaying Information About Environmental Objects

The new Environmentals List dialog box lists environmental objects (routes, waypoints, overlays, hazard clouds) that are published on the network. You can show all environmental objects or filter by a particular type.

You can also display information about individual environmental objects.

Fire/Detonate Option Separation

You can now configure the display of fire and detonate lines independently in the Display Options dialog box. The Options menu still has one control for both items. The menu option overrides the individual settings in the Display Options dialog box.

Enabling and Disabling the Display of Fire and Detonate Interactions

The Symbols tab on the Display Options dialog box now lets you enable and disable the display of fire and detonate interaction graphics. If these graphics are disabled, fire and detonation lines are also disabled.

Hazard Cloud Display

The GUI now displays hazard clouds that are generated using a protocol developed by ITT. The clouds can either be shown as ellipses with a radius (for each puff associated with a cloud) or as a bounding volume that encompasses the entire cloud. You can display an image instead of a colored ellipse; however, displaying images will dramatically decrease performance of the front-end.

The Hazard Options dialog box lets you configure the display of hazard clouds. These settings are saved to the *settings.mtl* file.

You can display information about a cloud as a whole and each individual puff that makes up a cloud. Since puffs can be densely packed, only one puff can be selected at any given time.

Shadow Text

A new display option has been added to allow shadowing of text for greater label clarity on symbols.

Ability to Specify A Symbol Map and Create Menu in Scenario Files

You can now specify the symbolmap file and create menu configuration file (alternative to *vrf.ent*) in a scenario file. You can specify them in the Open Database dialog box when you create a scenario, or you can edit the scenario file of an existing scenario. If the parameters are blank, the default files are loaded.

API Changes

This section describes changes to the VR-Forces APIs.

GUI API

GUI Remote Control

This section describes the classes that support remote control (for example, loading a database) of Plan View Displays or VR-Forces front-ends by sending MTL commands.

PVD Remote Controller

The *DtPvdRemoteController* class constructs and sends MTL commands over the network to be received by other Plan View Displays or VR-Forces front-ends, for example, Open Database. This class uses the *DtMtlNetCommandDispatcher* class to send and receive MTL style commands that can be parsed and executed by the *DtMtlNetParser*. It wraps the *DtMtlNetParser* class, shielding the caller from having to know the syntax of the underlying MTL commands. It also provides a convenient place to collect all of your MTL commands. *DtPvdRemoteController* implements the Open Database and Reflect commands. The chat example demonstrates how to extend the PVD controller to add a new MTL command.

DtMtlVariable

The *DtMtlVariable* class knows how to create and parse itself for all of the intrinsic MTL types (list, string, fixed and float). It has convenience member functions for adding objects (such as a vector) to a command argument. *DtMtlVariables* are used by *DtMtlCommands* to construct an MTL string that can be parsed locally or sent over the network for parsing by remote Plan View Display and VR-Forces GUIs. Typically, you will create *DtMtlVariable* objects when you are creating a list. To add a variable of a specific type, you use the appropriate *DtMtlCommand* addVariable() member functions, described in this section.

The *DtMtlVariable* object is a list of *DtMtlVariable* objects.

To create and add a list MTL variable to a *DtMtlCommand*:

1. Create a new *DtMtlVariable* object.
2. Call addElement() on this object to add the list elements.
3. Add the *DtMtlVariable* object to the desired *DtMtlCommand* object by passing it into the *DtMtlCommand* through its addVariable() member.

While the *DtMtlVariable* must be allocated by the caller, once it is added to a *DtMtlCommand* object (through the addVariable() call), the *DtMtlCommand* object assumes responsibility for deleting the *DtMtlVariable*.

DtMtlCommand

DtMtlCommand is an abstract base class that defines how an MTL command is created. Subclassing and adding member functions for getting and setting variables allows for greater type safety.

You must implement the following member functions in derived classes to fully define a MTL command:

- `virtual DtString command() const = 0` – A command string used when creating the MTL representation of the command.
- `virtual void setupVariableList() = 0` – Sets up the variables that will be placed into the command string. It is called from the commandString() member function of the base class. You can add variables to the list of variables that will be added to the command by using the addVariable() member function.

For each MTL command you want to add to an MTL command dispatcher, add it to the parser as follows:

```
myMtlParser.registerCommand(DtMyMtlCommand().command(), callbackFunc,
  usrPtr);
```

This registers your new command with your MTL dispatcher so it can be referenced in MTL commands and scripts. Failure to register your command with the dispatcher will result in the command being ignored.

When a callback is invoked, a list of lisp tokens is supplied in the first parameter and the lisp environment in the second:

```
void DtlObjPtr callbackFunc(const DtlObjPtr& args, const DtlObjPtr& env)
```

You can use these arguments in their pure form, or you can restore a command object by passing these arguments to an MTL command constructor. You must create a constructor of this type, calling the base class constructor, if you want to automatically rebuild your command. This can be done as follows:

```
DtMyMtlCommand::DtMyMtlCommand(const DtlObjPtr& args, const DtlObjPtr&
  env) :
DtMtlCommand(args, env, command())
{
  myInt = (int)variable(0);
  myString = (const char*)variable(1);
}
```

Part of this reconstruction will also be the assignment of the user pointer set when adding this member function. This can be referenced from the user() member function of the newly restored command. You can use this member function to call an internal member function from the static callback:

```
void DtlObjPtr callbackFunc(const DtlObjPtr& args, const DtlObjPtr& env)
{
  DtMyMtlCommand command(args, env);

  ((DtMyMtlCommand*)command->user())->process(command);
}
```

DtMtlCommandDispatcher

DtMtlCommandDispatcher is a convenience class that allows you to add registered commands to the MTL environment. It is a wrapper around the *DtMtlEnvironment* class that allows usage of the environment without having to know the lower level MTL calls.

DtMtlNetCommandDispatcher

DtMtlNetCommandDispatcher is a network enabled version of the command dispatcher that you can use to automatically register for network receipt of MTL commands and invoke registered callbacks when those messages are received. To register these commands, register the parser with the MTL command interaction as follows:

```
DtMtlNetCommandDispatcher parser;
parser.registerCommand(DtMyMtlCommand().command(), callbackFunc, usr);
DtMtlCommandInteraction::addParser(&exConn, &parser);
```

When an MTL command is received over the network, the callbackFunc registered will be invoked.

DtPvdMtlCommandDispatcher

DtPvdMtlCommandDispatcher checks to see if the user has decided to ignore messages received over the network.

Loading Script Files at Startup

The `-R` startup option lets you load a script file at startup. The script file can contain commands that have been registered with the default application parser. This parser can be retrieved from the commandDispatcher() member function in the *DtPvdAppEvent-Controller* object.

Object Transparency

A new class, *DtBaseSymbolDrawer,* has been added to support object transparency. If you have created your own drawer and want to take advantage of transparency drawing, instead of subclassing your drawer from *DtSymbolDrawer*, subclass from *DtBaseSymbolDrawer*. All other code can stay the same.

API Changes to Support Transmitters

The Connection Manager emits the following signals for transmitters:

```
transmitterAdded(const DtReflectedRadioTransmitter&)
transmitterRemoved(const DtReflectedRadioTransmitter&)
transmitterUpdated(const DtReflectedRadioTransmitter&)
```

It emits the following signal when a signal PDU is received:

```
signalInteraction(const DtSignalInteraction&);
```

When a new transmitter is added, an associated *DtTransmitterModelData* is created and a new model data object is added to the internal application model data list. This model data is associated with an owning entity model data. If no owning entity exists, the transmitter is not displayed. If the entity appears later in the simulation, the transmitter will be hooked up to that entity and displayed with that entity.

The symbol generator also emits the following signals when a signal PDU is received from the Connection Manager:

```
// Sent when a signal is first received and associated with a transmitter
signalReceived(const DtTransmitterModelData*,
  const DtSignalInteraction&);

// Sent when a transmitter has stopped transmitting after it had been
// transmitting
stoppedTransmitting(const DtTransmitterModelData*);
```

A new class *DtSignalMap* has been added. A *DtPvdApplication* object owns an instance of this class, and any time a new signal PDU is received it tries to map the signal to a possible sender of a signal on that same frequency. If no signal is matched, the signal is considered the first of that kind on that frequency and any subsequent signals received on that frequency (until a specified time has elapsed) are considered a match for that frequency and a communication link is made in the map. The display then draws a communication line based on this new information.

Symbol Mappers

The getColorForForce() and modifySymbol() member functions now take a `const DtEntityMapperKey&` as the last parameter.

This parameter will contain the entity type requesting the color.

Classed Removed

- The *DtPvdHelpMenu* has been removed and renamed to be a more generic *DtHelpMenu* included by *tmHelpMenu.h*. The basic functionality and menu items are the same.
- The *DtAboutBox* has been renamed to be *DtPvdAboutBox*. You must now include *pvdAboutBox.h* to override about box functionality.

Changes to the FED File

A new object has been added to the FED file. If you use a customized FED file, you must add the following to it:

```
(class PvdViewControl best_effort receive
  (parameter MtlCommandPdu)
)

(class MtlCommand best_effort receive
  (parameter MtlCommandPdu)
)
```

Object Parameters Database Changes

This section describes changes to the object parameters database.

- There was previously no way to specify the names of both the *DtAutomotivePSR* and the *DtVrfBalGunPSR* used by the *DtIndividualLifeformAquireComponent* through a component descriptor. This prevented the *DtIndividualLifeformAquire-Component* from being able to be configured with arbitrary process state repository names. To correct this problem, VR-Forces has a new type of component descriptor, *DtIndividualLifeformAcquireDescriptor*. It includes a new parameter for configuring the name of the *DtVrfBalGunPSR*. You can now configure a *DtIndividualLifeformAquireComponent* with both a *DtAutomotivePSR* and a *DtVrfBalGunPSR*, as follows:

```
(lifeform-acquire
   (component-descriptor-type "individual-lifeform-acquire-descriptor")
   (component-type  "individual-lifeform-acquire-controller")
   . . .
   (process-state-repository-name  "automotive-process-state-repository")
   (weapon-process-state-repository-name  "turreted-ballistic-gun-
  process-state-
repository")
   . . .
)
```

- Because of the changes to *DtBoundingVolume* (see "DtBoundingVolume Migration Guide," on page 32), a zero local bounding volume is no longer appropriate (it results in undefined behavior). As a result, all object parameters database file bounding volume values have been updated to have the default values, (1,1,1), for their **local-bvol** parameter, if that parameter was previously zeroes. Any entry with a zeroed **local-bvol** parameter will use the default values and ignore the zeroed bounding volume, and a warning will be generated. This is a temporary solution, until the bounding volume parameter can be removed from non-volumetric entries.
- Paths in *.ope* files are now relative to the file, not the application. See "Changes to Pathname Interpretation in Object Parameters Database Files," on page 12 for details.

Migrating Customized Object Parameters Databases

VR-Forces 3.8 has many changes and additions to the object parameters database, including the new path planning, damage, obstruction, and aggregate components. If you are using a customized object parameters database and want to take advantage of these new capabilities, you must merge your changes into the new format. This section describes an approach for migrating customized object parameters databases from previous versions of VR-Forces to the latest format.

Customized object parameters database are likely to be in one of two formats – the single-file format used before VR-Forces 3.7, or multi-file format introduced in VR-Forces 3.7. They are also likely to have two types of customizations – changes to the default parameters values supplied with VR-Forces, or the use of locally developed components.

Converting Pre-VR-Forces 3.7 Object Parameters Databases

To convert pre-VR-Forces 3.7 object parameters databases:

1. Run the utility *convertParamDb* to convert the file to the latest version. If you have added new kinds of descriptor or parameter classes, you must rebuild *convertParamDb* to incorporate your classes, and then run it to convert to the new format. See *VR-Forces 3.7 Release Notes*, *VR-Forces 3.7 Release Notes*, or *VR-Forces Migration Guide* for the procedures for using *convertParamDb*.

! The *convertParamDb* utility only converts the format of the object parameters database file. It does not add the new components introduced in VR-Forces 3.8.

---

2. Manually reconfigure the objects in the object parameters database to use the new VR-Forces 3.8 components.

If your customized file only changes parameter values, it might be easier to identify the changes and edit the default VR-Forces 3.8 object parameters database using the OPD Editor. If you have extended VR-Forces and added new components, you will have to evaluate the scope of your changes to determine if it is easier to convert the object parameters database and update the resulting files with VR-Forces 3.8 changes or start with the VR-Forces 3.8 object parameters database and add your extensions.

Merging Customized Object Parameters Databases that Use the Current Format

If you have a customized object parameters database that uses the current format, to take advantage of the changes introduced in this release, you must merge your object parameters database with the release version of the object parameters database. This will incorporate any changes we have made to the 3.8 version with those extensions you have made in previous releases.

To merge object parameters databases:

1. Identify the *.ope* files that you have customized. Compare them to the release versions using a text editor or a graphical comparison/merge tool such as Araxis Merge, or Beyond Compare. Merge your changes into the new file.
2. For customized *.ope* files that do not have comparable release versions, use the OPD Editor to add, delete, or edit components to incorporate the new VR-Forces features as desired.

Centralized Fire and Detonation Processing

Processing of damage and fire interactions has been centralized to simplify the process and improve performance. VR-Forces has a new actuator and descriptor and a new sensor and descriptor for damage adjudication and under-fire determination and two new classes for centralizing the processing, the *DtDetonationManager* and the *DtFireManager*.

Previously, when determining if an entity was under-fire or taking damage, every entity registered for notification of detonation and fire interactions. Then, when one occurred, each entity analyzed the interaction to see if it was applicable to them, and if so, to process it. This was very inefficient as most entities are not the target of a detonation or close enough to be influenced.

Centralizing the processing removes the need for every entity to process all interactions. It relies on the new manager objects to notify the appropriate entity or entities when an interaction involving them occurs. For some types of fire interactions, all entities must still be notified to determine if they are under fire. But for other types, this results in significant improvements.

The new components work together to minimize processing of interactions. The *DtFireManager* registers for both fire and detonation interactions, and the *DtDetonationManager* registers for detonation interactions. Thus the analysis of the interactions for relevance occurs in a single place. The new components – the *DtDamageAdjudicationActuator* and *DtUnderFireDeterminationSensor* register themselves and their entities, and their appropriate callback functions with the new managers. When a detonation or fire interaction occurs, the fire and detonation managers determine which registered entities are affected, and notify only those entities of the interaction. Thus only the appropriate registered entities need process the interaction. There are also the appropriate derived versions of these classes for individual lifeform entities.

The new under fire determination sensor has an added parameter in its descriptor: **under-fire-from-friendly**, which specifies whether or not an entity will consider itself under fire from friendly entities.

To maintain backwards compatibility, the existing components should be used, as they have been updated with bug fixes and the new **under-fire-from-friendly** capability. If you have a customized object parameters database and want to use the new classes, update your object parameters database files to use the new actuator and sensor: damage-adjudication-actuator and under-fire-determination-sensor respectively (or individual-lifeform-damage-adjudication-actuator for lifeform entities). If you have derived classes from the old damage actuator, either continue to use the original, or derive your components from these new classes.

The object parameters database files that ship with VR-Forces have been updated to use the new components.

DtBoundingVolume Migration Guide

The bounding volume class has been extensively refactored for several reasons:

- To fix several bugs in the design and implementation of the class.
- To provide an intuitive and correct interface to the class.
- To improve the general performance of both the class and of other areas of the simulation that used the class.

As such, the interface and usage of the class has changed significantly. This section describes the new interface for the *DtBoundingVolume* class as well as how to migrate existing code to the new interface. This section attempts to cover the most general usage of the *DtBoundingVolume* class, and thus does not document all changes. However, the class documentation has been extensively updated to explain the structure, interface, and usage of the class. For more detailed information, please look in the class documentation.

Construction

The constructor has changed to take only the necessary elements to create a *DtBoundingVolume* – the half length, width, and height of the lattice, and potentially the length, width and height of the lattice offset as well, in either vector form or component form. Additionally, a *DtBoundingVolume* can now be constructed or assigned from another *DtBoundingVolume* instance.

The *DtDcm* class is no longer used with regards to the *DtBoundingVolume* class. If you have code that is initializing the bounding volume using a *DtDcm*, pull out the diagonal components of the *DtDcm* and use them individually as the parameters for the constructor. Better yet, refactor the code to remove the need for the *DtDcm* at all, and simply pass around the construction parameters directly, for example:

Previously:

```
DtDcm entityDcm;
const DtBoundingVolume* entityBvol = entity()->vrfState()-
  >boundingVolume();
entityBvol->GetLocalBoundingVolume(entityDcm);

DtBoundingVolume* tempBoundingVolume = 0;
tempBoundingVolume = new DtBoundingVolume(entityDcm, entityBvol-
  >GetBvOriginOffset());
```

Now:

```
const DtBoundingVolume& entityBvol = entity()->vrfState()-
  >vrfObjectGeometry()->localBoundingGeometry().boundingVolume();

DtBoundingVolume* tempBoundingVolume = 0;
tempBoundingVolume = new DtBoundingVolume(entityBvol);
```

Alternatively, when setting the entire bounding volume, for initialization, and so on, the operation can now be done as follows:

Previously:

```
// Set the bounding volume
boundingGeometry.setBoundingVolume(boundingVolumeDiagonal);
```

Now:

```
// Set half dimensions.
boundingGeometry.setHalfLengthWidthHeight( boundingVolumeDiagonal[0],
  boundingVolumeDiagonal[1], boundingVolumeDiagonal[2]);
```

Accessing Properties

The bounding volume now contains only its essential properties – the half width, height, and length of the lattice and the lattice offset. It no longer contains a position, a world bounding volume, a position or origin, or an orientation. Nor does it store the results of intersection tests, as these have now been moved to a separate intersector class, if not removed entirely. Later in this section, there are examples of how properties of the previous bounding volume were accessed, and how they are accessed now

Getting the Bounding Volume

The previous bounding volume class held two bounding volumes, a local bounding volume where local was not database coordinates, but lattice coordinates, and a transformed (database) bounding volume. These were both accessed in the same manner:

```
DtDcmRef worldBoundingVolume;
entityBoundingVolume.GetBoundingVolume(worldBoundingVolume);

DtDcmRef localBoundingVolume;
entityBoundingVolume.GetLocalBoundingVolume(localBoundingVolume);
```

The new bounding volume class is the lattice coordinate system bounding volume. So instead of presenting an interface for retrieving the entire bounding volume, it presents a more appropriate interface to the half-height, half-width, and half-length. Note that almost always, when developers accessed the bounding volume, they were actually trying to access the components of that bounding volume.

So to access the dimensions of the bounding volume, or to get the offset, call the appropriate member:

```
double length = entityBoundingVolume.length();
double halfWidth = entityBoundingVolume.halfWidth();
```

A more important change to the interface, for mutating the bounding volume, is documented in the next section.

Changing the *DtBoundingVolume* Class (Removal of the mutate() function):

The mutate() function was removed in favor of a simpler and more correct method for altering the structure of a bounding volume. It has been replaced by the stretchForward() and stretchBackward() functions. These functions alter the bounding volume in the expected manner.

For example, the collision detector used to mutate the collision bounding volume in order to stretch it out to detect potential collisions in front of the entity:

```
DtDcm entityLocalBV;
entity()->vrfState()->boundingVolume()>
  GetLocalBoundingVolume(entityLocalBV);

double oldRadiusInX = entityLocalBV [0][0];

double stoppingDistance = 1.0;
stoppingDistance = groundAutoController->
  calculateCurrentBrakingDistance();

double newRadiusInX = 0.0;
newRadiusInX = (stoppingDistance * stoppingDistanceFactor()) +
  oldRadiusInX;

entityLocalBV [0][0] = newRadiusInX;
entityLocalBV [1][1] += obstructionSensorDesc->lateralClearance();

DtVector bvolOffset;
bvolOffset[0] = newRadiusInX - oldRadiusInX;
bvolOffset[1] = 0.;
```

```
bvolOffset[2] = 0.;

myCollisionBV->Mutate(entityLocalBV, newRadiusInX, bvolOffset,
  bvolOffset[0]);
```

Now, the collision detector determines by what amounts it should stretch the bounding volume, and then does so. Note that it now has to reset the bounding volume to its initial state, or the stretch operations will be compounded.

```
// Make sure to reset the collision bounding volume to the default (actual
// entity local bounding volume) before we stretch it
const DtBoundingVolume& entityBvol = entity()->vrfState()->
vrfObjectGeometry()->localBoundingGeometry().boundingVolume();
*myCollisionBV = entityBvol;

// modify our local bounding volume in the x dimension by adding braking
// distance. The x dimension should be forward, so if we're ever not
// moving in the forward direction then this is incorrect. (such as for
// helicopters, submarines, etc.)  Note that reverse is also OK, since it
// is still along the local (body) x axis.
double stoppingDistance = 1.0;
calculateStoppingDistance(*obstructionSensorDesc, stoppingDistance);

double lengthStretch = stoppingDistance * obstructionSensorDesc->
  stoppingDistanceFactor();
double widthStretch = obstructionSensorDesc->lateralClearance();

if(entity()->vrfState()->currentSpeed() >= 0)
{
  myCollisionBV->stretchForward( lengthStretch, widthStretch );
}
else
{
  myCollisionBV->stretchBackward( lengthStretch, widthStretch );
}
```

General Use of Bounding Volumes

Most uses of the bounding volume, other than for its dimensions, involve a bounding volume and its origin in local (database) coordinates. For many reasons, this data was removed from the *DtBoundingVolume* class. The following code constructs it, given a bounding volume, a transformation matrix, and a local location.

```
DtBoundingVolume bv;
DtVector bvPosition; // local position
DtDcm latticeToRef;  // transformation from lattice to reference CS.

DtDcm worldBV;

// Create a diagonal matrix from the vector.
DtDcm tempMatrix;
tempMatrix[0][0] = bv.halfLength();
tempMatrix[1][1] = bv.halfWidth();
tempMatrix[2][2] = bv.halfLength();

// Rotate the world bounding volume in world coordinates.
DtDcmDcmMul(latticeToRef, tempMatrix, worldBV);

// And now calculate the reference origin
```

```
DtVector worldOrigin;
if(bv.isOffset())
{
  DtVector worldOffset;
  DtDcmVecMul(latticeToRef, bv.latticeOffset(), worldOffset);
  DtVecAdd(bvPosition, worldOffset, worldOrigin);
}
else
{
  worldOrigin = bvPosition;
}
```

The following example, taken from source code, converts the bounding volume from a lattice coordinate system to a transformed coordinate system, in this case the local coordinate system. To calculate an extent from an entity's bounding volume, the local (database) bounding volume must be calculated, and then used to expand an extent so that we have the smallest axis-aligned bounding box (what our *DtExtent* class is) that bounds the transformed bounding volume.

(This is the actual implementation of the function for expanding an extent to hold a bounding volume.)

```
void DtExpandExtentToIncludeBoundingVolume(DtExtent& extent,
  const DtBoundingVolume& boundingVolume, const DtDcm& worldOrientation,
  const DtPoint& worldLocation)
{
  DtVector vecWorldLocation(worldLocation.x(), worldLocation.y(),
    worldLocation.z());

  double halfLength = boundingVolume.halfLength();
  double halfWidth = boundingVolume.halfWidth();
  double halfHeight = boundingVolume.halfHeight();

  // First determine the local extent, in the lattice CS
  std::vector<DtVector> localExtentPoints(8);
  // lower, back, left corner
  localExtentPoints[0][0] -= halfLength;
  localExtentPoints[0][1] -= halfWidth;
  localExtentPoints[0][2] -= halfHeight;

  // lower, back, right corner
  localExtentPoints[1][0] -= halfLength;
  localExtentPoints[1][1] += halfWidth;
  localExtentPoints[1][2] -= halfHeight;

  // lower, front, left corner
  localExtentPoints[2][0] += halfLength;
  localExtentPoints[2][1] -= halfWidth;
  localExtentPoints[2][2] -= halfHeight;

  // lower, front, right corner
  localExtentPoints[3][0] += halfLength;
  localExtentPoints[3][1] += halfWidth;
  localExtentPoints[3][2] -= halfHeight;

  // upper, back, left corner
  localExtentPoints[4][0] -= halfLength;
  localExtentPoints[4][1] -= halfWidth;
  localExtentPoints[4][2] += halfHeight;
```

```
    // upper, back, right corner
    localExtentPoints[5][0] -= halfLength;
    localExtentPoints[5][1] += halfWidth;
    localExtentPoints[5][2] += halfHeight;

    // upper, front, left corner
    localExtentPoints[6][0] += halfLength;
    localExtentPoints[6][1] -= halfWidth;
    localExtentPoints[6][2] += halfHeight;

    // upper, front, right corner
    localExtentPoints[7][0] += halfLength;
    localExtentPoints[7][1] += halfWidth;
    localExtentPoints[7][2] += halfHeight;

    if (boundingVolume.isOffset())
    {
      // Make sure to account for the lattice offset if it exists
      for (int pointIndex = 0; pointIndex < 8; ++pointIndex)
      {
        DtVecAdd(localExtentPoints[pointIndex],
          boundingVolume.latticeOffset(), localExtentPoints[pointIndex]);
      }
    }

    std::vector<DtVector> worldExtentPoints(8);
    for (int pointIndex = 0; pointIndex < 8; ++pointIndex)
    {
      DtDcmVecMul(worldOrientation, localExtentPoints[pointIndex],
        worldExtentPoints[pointIndex]);

      DtVecAdd(worldExtentPoints[pointIndex], vecWorldLocation,
        worldExtentPoints[pointIndex]);

      extent.expandToInclude(worldExtentPoints[pointIndex]);
    }

    return;
  };
```

For more information, see the *DtBoundingVolume* and *DtBoundingVolumeIntersector* class documentation, as they contain extensive documentation of these concepts.

Changes to the *DtCgf* Class

The *DtCgf* class has been modified in the following ways:

- Access to the terrain database member has been restricted to accessor functions so that you can easily override the type of *DtTerrainDatabase* used by the application. You now set or get the terrain database through the accessors (the member itself is now private).
- You can now create a derived DtTerrainDatabase and replace the one shipped with VR-Forces. The *DtCgf* must now keep track of the vrfCreator object. *DtCgf* does not maintain ownership of user specified vrfCreator objects, but will do so for instances it creates. This vrfCreator is used to create the terrain database. Thus by registering a new create function for a derived *DtTerrainDatabase* with the vrfCreator and then by supplying that creator as a parameter to the DtCgf::init() function, you can specify that your derived *DtTerrainDatabase* be used.

  The *DtVrfCreator* and related classes have been updated with the ability to register or use creation functions for the *DtTerrainDatabases* and to create instances of them.
- The *DtFireManager* and *DtDetonationManager* have been added. They are responsible for processing fire and detonation interactions, and for working with the *DtDamageAdjudicationActuator* and the *DtUnderFireDeterminationSensor* for adjudication damage and determining under-fire status. The *DtCgf* class owns these objects and is responsible for their construction and destruction. See "Centralized Fire and Detonation Processing," on page 31.
- New functions for logging frame rate statistics have been added. These are purely additive changes, and do not affect backwards compatibility at all.
- The copy and assignment operators were made explicitly protected and not implemented to guarantee that the compiler did not generate them. (This was assumed true and intended before.)

Explicit Hiding of Functions

Many classes in the API were hiding functions from the classes they derived from. In most cases, this was intentional. However, to ensure that developers know the hiding is intentional, and to remove any potential compiler (or other tool) warnings, many of the hidden functions were explicitly hidden by declaring them as in use, but private. For example, in the *DtVrfObjectManager* class, the following was added:

```
private:
  // Intentionally hiding these functions
  // They have been replaced by new declarations/definitions in this
  // class.
  using DtRwList::add;
  using DtRwList::remove;
```

DtSpecificationAttribute Changed to Use Symbol Strings

The *DtSpecificationAttribute* class has been updated to contain a symbol string instead of an actual string. This significantly reduces the amount of memory needed to store large numbers of specification attributes in the specifications. Most specifications contain the same attributes, and so this reduction is significant for terrain files with large amounts of vector data. You should not have to do anything to update your code. Additionally, the label member has been made private, but with public accessors so that derived classes can easily override the functionality to store the label another way. For more information, see "Symbol Strings," on page 47.

Spatial Organization of Objects

The *DtVrfObjectManager* class now contains a *DtSpatialVrfObjectManager,* which it uses to spatially organize specified *DtVrfObjects*. A filter function predicate is used to determine which *DtVrfObjects* are spatially sorted, in order to allow you to easily change the sorting in a derived class. Additionally, the creation, initialization, and use of the *DtSpatialVrfObjectManager* class is all performed in the appropriate functions, allowing for user modification. For a complete description of spatial subdivisions, see "Spatial Subdivisions," on page 41.

A simple algorithm is used to determine the spatial organization configuration (number of subdivisions in each dimension). Also, all *DtVrfObjects* are stored in the Spatial Manager, including routes, waypoints, and so on.

To access the spatially sorted vrfObjects, the getVrfObjects() functions are provided to get all vrfObjects that may intersect the specified primitive (currently only *DtChord* or *DtExtent*). The resulting set contains all objects that may intersect the chord or extent. Not all are guaranteed to do so, but the set will not be missing any that actually do. If the exact intersection is required, as for line-of-sight testing in the vrfObjectStateRepository, then further refinement is required.

Additionally, the functions, DtRwList::add and DtRwList::remove, that were being hidden implicitly have been made hidden explicitly. This does not change the API, but makes it clear that the functions are being hidden on purpose.

For more details on how to override the spatial organization behavior in both this class and of the vector obstructions, see the appropriate sections in the manual.

Changes to *DtVrfObjectStateRepository*

The *DtVrfObjectStateRepository* now uses the spatially managed vrfObjects and vector obstructions when determining line-of-sight between entities. The existing function, intersectsEntity(), now calls a new function, intersectEntities(), to see if the specified chord intersects any entities other than the entity being tested for LOS. As a note, the same predicate used for determining if an entity is to be tested for LOS is used to screen out entities from the spatially sorted intersection results.

Changes to *DtTerrainDatabase*

The *DtTerrainDatabase* class now uses a *DtSpatailSubdivision* for spatially organizing the terrain, instead of a *DtSpatialIndex*. The *DtSpatialIndex* remains in the library, and is used for verification purposes against the *DtSpatialSubdivision*, but it is no longer supported for other use.

The accessor functions for the terrain database should change slightly. For example, to request the spatial subdivision, instead of saying:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = terrainDb.spatialIndex();
```

the following is used:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = terrainDb.spatialSubdivision();
```

The reason the name was changed explicitly, instead of just changing what the spatial index was, was to make sure that there was no confusion between the old uniform spatial container for the terrain, and the new generic uniform spatial container, which provides a minimal interface. See the details of the *DtSpatialSubdivision* for more information.

Additionally, the following were all updated:

- The version of the GDB format was incremented, specifically to account for the addition of the spatial subdivision.
- transformCoordinateSystem() now takes an optional parameter that allows the existing coordinate system to be retransformed to itself. This is useful for reprojecting a coordinate system to a new origin.
- New functions for finding the ground height and the intersection at that location have been provided: See terrainHeightAndIntersection() in the class documentation.
- Support for the Flat Earth coordinate system projection has been added, as well as functions for transforming to it and from it. Functions for transforming from UTM to UTM have also been added, which are to be used to reposition an existing coordinate system.
- The memory manager member has been made mutable, as it is modified by const accessor functions.

Spatial Subdivisions

The *DtSpatialSubdivision* class represents a uniform spatial container. The container is divided up into a lattice of cells, with resolution specified at creation. Each cell is a container of the objects in it. Its purpose is to efficiently store and search objects spatially. In this regard it is the same as the original *DtSpatialIndex* class. However, spatial subdivisions are more flexible and extensible.

The *DtSpatialSubdivision* is a templated container, similar to the std::list or std::vector classes. As such, it can hold any specified item. The intended use is to store elements that have "spatial" properties – that is elements that have an actual volume, position, orientation, and so on. Because of this flexibility, not only is the terrain spatially sorted in VR-Forces 3.8, but so are the *DtVrfObjects* and the *DtVectorObstructions*, resulting in a significant improviement in performance when simulating large numbers of entities.

The responsibility for and intelligence necessary to perform operations on the spatial subdivision, such as intersection with primitives, insertion of elements into, and so on, have been separated out into specific classes for the distinct operations. This greatly facilitates extending existing capabilities and adding new ones. As a result, the spatial subdivision functionality is spread out across an entire family of classes, as described in this section.

Spatial Subdivision Container

The *DtSpatialSubdivision* is the container class. It is defined in *spatialSubdivision.h*, in the geometry library.

Spatial Subdivision Intersectors

The spatial subdivision no longer contains the ability to intersect geometric primitives and other objects with itself. The ability to do that has been moved into a set of intersector classes. These intersector classes know how to intersect a primitive and a spatial subdivision, and apply a specified operation to the results of the intersection through a functor interface.

The functor interface allows for tremendous flexibility in the type and complexity of operations that are performed on the spatial subdivision. By changing the functor supplied to the intersect() member function, different operations can be performed, such as inserting elements into the spatial subdivision, gathering elements of specific characteristics from the spatial subdivision, or simply accumulating all cells in the spatial subdivision that are part of the intersection. The following are examples of intersectors:

- *DtSsChordIntersector* (*ssChordIntersector.h*) – This class intersects a specified chord and spatial subdivision, applying a functor to the intersection results.
- *DtSsExtentIntersector* (*ssExtentIntersector.h*) – This class intersects a specified extent and spatial subdivision, applying a functor to the intersection results. Note that this class also provides a function for determining the min/max intersected cell indices, which can sometimes be more useful than the actual full intersection call.

Spatial Subdivision Functors

The spatial subdivision functors are simple classes that are used by the spatial subdivision intersectors to perform different operations on the results of intersection tests. These operations include insertion of elements into spatial subdivision cells (*DtSsInsertionFunctor*), accumulation of spatial subdivision cells for later analysis (*DtSsAccumulationFunctor*), among others.

While simple, these functors differ from standard functors in the following important ways:

- The functors are passed by *reference*, not by value. They do not have to maintain state, but they are expected to.
- The functors return a Boolean to indicate whether or not they are done processing inputs, and *not* whether or not an error occurred. This is to allow for efficient processing of intersection results. For instance if a functor is looking for any intersection of a chord and a polygon, it can stop processing after the first intersection is found, simply by returning true. To determine if an error occurred, the functor *must *either maintain internal error state or use exceptions.

The following are examples of functors:

- *DtSsFunctor* (*ssFunctor.h*) – This is the base class for all spatial subdivision functors. It does not perform any operation.
- *DtSsAccumulationFunctor* (*ssAccumulationFunctor.h*) – *DtSsAccumulationFunctor* contains a container, which it uses to store elements passed in to it's function operator (see the function definition for more details). Thus it can be used to accumulate the set of cells comprising the intersection of a primitive with the spatial subdivision.
- *DtSsInsertionFunctor* (*ssInsertionFunctor.h*) – *DtSsInsertionFunctor* is used to insert one object into another, for instance in inserting elements into the spatial subdivision cells.

Spatial Subdivision Inserters

The spatial subdivision inserter classes are utility classes that combine the appropriate intersector with an insertion functor, in order to facilitate inserting of elements of the appropriate type into a spatial subdivision. *DtSsGdbNodeInserter* (*ssGdbNodeInserter.h* and *ssGdbNodeInserter.cxx*) is an example of an inserter. It inserts *DtGdbNodes* into a spatial subdivision. Generally speaking, it uses an extent intersector to find the cells which the *DtGdbNode* intersects, and then an insertion functor to insert the *DtGdbNode* into those cells.

Migrating from *DtSpatialIndex* to *DtSpatialSubdivision*

This section explains how to migrate existing code from using the *DtSpatialIndex* to using the new *DtSpatialSubdivision*. For more information, please consult the class documentation.

Creating the Container

Creation has not really changed, except for the specification of the template parameter.

Previously:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = new DtSpatialIndex(params);
```

Now:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = new DtSpatialSubdivision<DtGdbNode*>(params);
```

Notes

- To change the type of contained element, change the template parameter – DtGdbnode*
- The parameters to the constructor functions have not changed.

Accessing the Container from the Terrain Database

The spatial subdivision is accessed in the same way as the spatial index, except that the function names have been changed to be spatialSubdivision.

Previously:

```
DtSpatialIndex* spatialIndex = 0;
spatialIndex = terrainDatabase->spatialIndex();
```

Now:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = terrainDatabase->spatialSubdivision();
```

Inserting Elements into the Container

Previously, to insert a node into the spatial index, the addNode() function was called. For example, when the terrain database created the spatial index for the terrain, it used the following code, where mySpatialIndex was the terrain database's spatial index and myTerrain was the terrain database's root terrain *DtGdbNode**:

```
mySpatialIndex->addNode(myTerrain);
```

However, using the new spatial subdivision, an inserter class must be used, because the spatial subdivision no longer knows how to insert other objects into itself.

The following example show how the terrain database now creates the spatial subdivision for the terrain polygons. The terrain() function returns the root terrain polygon node (*DtGdbNode**).

```
DtSsGdbNodeInserter terrainInserter;
if (!terrainInserter.insert(terrain(), *myTerrainSpatialSubdivision))
{
  error handling code...
}
```

Finding Terrain Intersections with the Terrain's Spatial Container

Previously, the spatial index provided a set of intersect functions that matched the intersect functions of the terrain database. These all took chords or bundles of chords as parameters, as follows:

```
spatialIndex->intersect(chord, intersectionPoint, intersectionTime);
```

Now, there is a helper class for finding terrain intersections, called *DtTerrainIntersector* (*terrainIntersector.h*). The terrain intersector contains all the knowledge of the group of different terrain intersection functions the spatial index supported, and in turn calls the appropriate intersectors, functors, and performs the appropriate analysis on the results. To find terrain intersections, do the following:

```
DtTerrainIntersector terrainIntersector;
terrainIntersector.intersect(chord, spatialSubdivision,
  intersectionPoint, intersectionTime);
```

Intersecting DtChords and DtExtents Directly with the Container

Previously, there was no way to explicitly intersect an extent with the spatial index, from a mathematical point of view, and get the resulting cells that made up the intersection - the logic was locked inside the spatial index class! Now the *DtSsChordIntersector* class (*ssChordIntersector.h*) contains all the intelligence for intersecting a *DtChord* and a spatial subdivision. To intersect a chord with an spatial subdivision, create an instance of a *DtSsChordIntersector*, and call its intersect function with the desired chord, spatial subdivision, and functor. See the class documentation for more information.

The following code is an abridged example of how an accumulation functor is used and then analyzed (some data has been omitted for brevity):

```
DtSsChordIntersector<DtGdbNode*> chordIntersector;
DtSsAccumulationFunctor<DtChord,
  DtSpatialSubdivision<DtGdbNode*>::DtSpatialSubCellType>
  accumulationFunctor;

if (chordIntersector.intersect(chord, spatialSubdivision,
  accumulationFunctor))
{
  // walk the accumulated cells, and analyze them appropriately
  DtAccumulationContainer::iterator cellIter =
    accumulationFunctor.container().begin();
  DtAccumulationContainer::iterator endCellIter
    accumulationFunctor.container().end();
  for (; cellIter != endCellIter; ++cellIter)
  {
    DtSpatialSubCellType* cell = *cellIter;
    DtSpatialSubCellType::iterator cellContentsIter = cell->begin();
    DtSpatialSubCellType::iterator endCellContentsIter = cell->end();

    for (;cellContentsIter != endCellContentsIter; ++ cellContentsIter)
    {
      // PERFORM ANALYSIS
    }
  }
}
```

Intersecting DtExtents with the Container

As with *DtChords*, previously there was no way to explicitly intersect an extent mathematically with the spatial index, and get the set of cells that comprised the intersection - the logic was locked inside the spatial index class. Now, the *DtSsExtentIntersector* (*ssExtentIntersector.h*) contains all the intelligence for intersecting a *DtExtent* and a spatial subdivision. To intersect an extent with an spatial subdivision, create an instance of an *DtSsExtentIntersector* and call its intersect function with the desired extent, spatial subdivision, and functor. See the class documentation for more information.

Extents are the only non-chord primitive for which intersectors are provided. While other intersectors exist for *DtVrfObjects*, *DtVectorObstructions*, and *DtGdbNodes*, all create extents from the base objects that are then used to intersect the spatial subdivision.

```
DtSsExtentIntersector<DtVectorObstruction*> extentIntersector;
if (extentIntersector.intersect(vectorObstruction, spatialSubdivision,
  functor))
{
  // do whatever you want to do when there is an intersection
}
else
{
  // do whatever you want to do when there isn't an intersection
}
```

Extending the *DtSpatialSubdivision* Classes

Because the spatial subdivision is a templated container, creating a spatial subdivision of a new type of element is as easy as altering the declaration.

For example, the following creates a spatial subdivision of pointers to *DtGdbNodes*:

```
new DtSpatialSubdivision<DtGdbNode*> gdbNodeSpatialSub;
```

The following creates a spatial subdivision of pointers to *DtVectorObstructions*:

```
new DtSpatialSubdivision<DtVectorObstruction*>
  vectorObstructionSpatialSub;
```

Creating A New Type of DtSsfunctor

Derive the new functor from the *DtSsFunctor* class. To use this functor, pass it as a parameter to an existing spatial subdivision intersector or other utility class, or to a newly created spatial subdivision utility class.

! Be sure that the new functor adheres to the interface of the *DtSsFunctor* classes.

Creating A New Intersector

Creating new intersector classes is probably one of the most useful extensions to the spatial subdivision classes. For instance, an intersector class for spheres or cylinders or other implicit surfaces would enable efficient selection of all elements within a specified radius of a location (or line).

To create a new intersector class, there is no need to derive from any parent class. The intersector class merely needs to know how to intersect the specified primitive with the spatial subdivision and to allow user-specified processing of the results.

So, to create a sphere intersector you could test every cell for inclusion with the specified sphere, using the properties of the spatial subdivision (cell deltas, location, and so on) to calculate the cells to test, and to pass these cells to the user-specified functor. This would be useful when selecting all cells within a blast radius. To take it a step further, a new functor could also be derived that would know how to intersect all the elements of a cell, such as *DtGdbNodes*, with the specified primitive. Used together, the new sphere intersector and sphere intersection functor could be used to select all polygons within the blast radius of a detonation.

Symbol Strings

The *DtSymbolString* class, and related classes, provide a way to treat strings as symbols instead of as ordinary strings. When a symbol string is created from a string, the string is stored inside of a table of strings and the symbol string is given a key with which to get the associated string from the table. As a result, unique strings are stored only once. This can result in significant memory savings if many instances of the same string were formerly stored in the system. Additionally, because each symbol string references its string by a key, string comparison becomes key comparison (much more efficient). In general, the symbol string classes are used in the reader/writer classes and the *DtSpecification* classes. You should not have to update code to use the *DtSymbolStrings*, other than to provide the appropriate functions if any classes were derived from reader/writer classes implementing member functions using *DtSymbolStrings*.

The symbol string classes and header files, which are in *gdb.lib*, are:

- *DtSymbolStore* (*symbolStore.h*) – The repository of unique symbols. Used, as a singleton, by the symbol string class.
- *DtSymbolString* (*symStr.h*) – The core symbol string class.
- *DtCStringHashKey* (*cstringHashKey.h*) – A string hash key based upon symbol strings, instead of full strings.

To maintain backwards compatibility, you do not need to make any changes to your object parameters database files. To use the new classes, update the object parameters database files to use the damage-adjudication-actuator and under-fire-determination-sensor.

New Classes for Managing Aggregates

The following sections describe new classes that manage aggregate behaviors.

New Aggregate State Repository Class

DtVrfAggregateStateRepository (*vrfAggregateSR.h*) is a state repository class for all aggregate entities. It maintains the current type of formation, and the current state of the formation (which subordinate is in which formation position). All aggregate entities are configured with this type of state repository.

New Aggregate Process State Repository Classes

- *DtAggregateMoveAlongPSR* (*aggregateMoveAlongPSR.h*) contains the state associated with movement along a route task. It is used by the *DtAggregateMoveAlongController* class.
- *DtAggregateMoveToPSR* (*aggregateMoveToPSR.h*) contains the state associated with a move to task. It is used by the *DtAggregateMoveToController* class.
- *DtAggregatePatrolAlongPSR* (*aggregatePatrolAlongPSR.h*) contains the state associated with a patrol along route task. It is used by the *DtAggregatePatrolAlongController* class.
- *DtAggregatePatrolBetweenPSR* (*aggregatePatrolBetweenPSR.h*) holds state associated with a patrol between task. It is used by the *DtAggregatePatrolBetweenController* class.

New Aggregate Controller Component Classes:

- *DtAggregateFormationController* (*aggregateFormationController.h*) manages the state associated with an aggregate entity's formation. This includes determining who is eligible to participate in the formation and who should be positioned relative to whom to form the formation. It also handles all *DtSetDataRequests* that affect formation, including *DtSetLocationRequests*, *DtSetHeadingRequests*, and *DtSnapIntoFormationRequests*. It sets the state in the aggregate entity's *DtFormationState* object (owned by the *DtVrfAggregateStateRepository* class).
- *DtAggregateMoveAlongController* (*aggregateMoveAlongController.h*) models movement of an aggregate along a given route. It registers for *DtMoveAlongTask* messages. Execution of the behavior is broken down into two phases. First, it issues a *DtMoveIntoFormation* subtask to itself (handled by the *DtMoveIntoFormationController*) to move the aggregate into position at the start of the route, in the given formation, and oriented in the correct direction. When this subtask is complete, it generates a separate 'working route' for each subordinate to follow, and sends a *DtMoveAlongTask* to each subordinate with its route. The 'working routes' generated are created at the offset relative to the original route necessary to maintain the proper lateral position of the subordinate in formation. This controller periodically monitors how well each subordinate is maintaining its position in formation and issues Set Speed Requests to speed up or slow down a given subordinate to try to maintain the correct front-to-back shape of the formation. The task is complete when all subordinates have sent a *DtTaskComplete* report message to the aggregate indicating they have reached the end of their respective 'working routes'.
- *DtAggregateMoveIntoFormationController* (*aggregateMoveIntoFormationController.h*) models movement of an aggregate into formation at a given position and heading. It registers for *DtMoveIntoFormationTask* messages.
- *DtAggregateMoveToController* (*aggregateMoveToController.h*) models movement of an aggregate to a given waypoint. It registers for *DtMoveToTask* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself.

- *DtAggregateMoveToLocationController* (*aggregateMoveToLocationController.h*) models movement of an aggregate to a given waypoint. It registers for *DtMoveToLocation* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself.
- *DtAggregatePatrolAlongController* (*aggregatePatrolAlongController.h*) models patrol along a route behavior. It registers for *DtPatrolRoute* messages. Its behavior is implemented by alternating between two subtasks: *DtMoveAlongTask* and *DtReverseDirectionTask*. When it first receives a *DtPatrolAlongTask*, it issues a *DtMoveAlongTask* subtask to itself. When this task is complete (it receives a *DtTaskComplete* report from itself), it issues a *DtMoveAlongTask* to itself in the reverse direction. It repeats the process indefinitely, alternating directions.
- *DtAggregatePatrolBetweenController* (*aggregatePatrolBetweenController.h*) models patrol between two waypoints behavior. It registers for *DtPatrolBetween* messages. Its behavior is implemented by sending *DtMoveTo* subtasks to itself, alternating between the two waypoints indefinitely.
- *DtAggregateTurnToHeadingController* (*aggregateTurnToHeadingController.h*) models behavior of turning to a given heading. It registers for *DtTurnToHeading* messages. It implements the behavior by sending a *DtMoveIntoFormation* subtask to itself at its current location, current formation, and new heading.

New Task Class

- *DtMoveIntoFormationTask* (*moveIntoFormationTask.h*) tasks an aggregate to move into a given formation, at a given location and heading.

New Set Data Request Classes

- *DtSnapIntoFormationRequest* (*snapIntoFormationRequest.h*) requests that an aggregate instantly reposition and reorient all of its subordinates into a given formation, at a given position and heading.
- *DtSetFormationTypeSetDataRequestType* (*setFormationType.h*) requests that an aggregate set its formation type to a given formation enumeration. It does not change the relative positioning of subordinates in formation.

Sending Task Messages

Aggregate controller components are now responsible for sending clear task messages to the subordinates to which they assign tasks. This was formerly done by the *DtPseudoAggregateTaskManager* (now obsolete). This was moved to aggregate controllers because that is where the logic for issuing tasks to subordinates resides.

Obsolete Classes

The following classes have been removed from the VR-Forces API:

- *DtPseudoAggregateFormation* (*psdAggFrmCtlr.h*)
- *DtPseudoAggregateMoveController* (*psdAggMvCtlr.h*).
- *DtPseudoAggregateTaskManager* is obsolete. Aggregates should now be configured with the same *DtLocalTaskManager* class as individual entities. *DtPseudoAggregateTaskmanager* has been typedef'd to *DtLocalTaskManager* for backward's compatibility.

Ability to Extend State Repositories at the Base Class Level

You can now extend the base state repository class *DtVrfObjectStateRepository* (*vrfObjSR.h*), as an alternative to, or in addition to extending leaf-level classes in the *DtVrfObjectStateRepository* hierarchy. The base state repository class, *DtVrfObjectStateRepository* now has-a *DtVrfStateRepositoryUserExtension* (*vrfSRUserExt.h*) that you can extend for all types of entities and objects.

For example, suppose you want to extend all types of entities and objects in VR-Forces (fixed-wing, rotary-wing, ground, and so on) with a new kind of identifier. Rather than extending each type of state repository to include the new identifier, you just need to create a single derived kind of *DtVrfStateRepositoryUserExtension* class. This allows you to write the extension once, and configure it as part of a variety of different kinds of entities, control objects, or overlay objects. The alternative to this approach would be to subclass each kind of state repository separately, extending each class in exactly the same way.

Like the *DtVrfObjectStateRepository* classes, classes derived from *DtVrfStateRepositoryUserExtension* are readable and writeable. Any *DtReaderWriter* data members you add to your derived *DtVrfStateRepositoryUserExtension* will be saved to an order of battle file along with the rest of an object's *DtVrfObjectStateRepository* data.

For complete details, see "Extending State Repositories at the Base Class Level.", in *VR-Forces Developer's Guide*.

Terrain API Changes

This section summarizes changes to the Terrain API.

GeometryNative Files Removed

The geometryNative project and files that were deprecated in the 3.6 release have finally been removed. Any remaining references to them should be changed to use the appropriate files in the Geometry project.

DtSpecificationAttribute Refactored

The *DtSpecificationAttribute* class has been updated to contain a symbol string instead of an actual string. This significantly reduces the amount of memory needed to store large numbers of specifications attributes in the specifications. Most specifications contain the same attributes, and so this reduction is non-trivial for terrain files with large amounts of vector data. You should not have to do anything to update your code. Additionally, the label member has been made private, but with public accessors so that derived classes can easily override the functionality to store the label another way.

New GDB File Version

The GDB file format has been incremented to account for the spatial subdivision changes. VR-Forces can read GDB files using earlier formats, but previous versions of the VR-Forces cannot read GDB files created with the release 2.83.8 tdbtool.

The new GDB version no longer explicitly stores the entire spatial subdivision in the file. It stores only the configuration, so that it can be recreated every time the file is imported. This only marginally affects the performance of importing GDB files, and drastically reduces versioning issues when reading GDB files.

Miscellaneous VR-Forces API Changes

The following changes were made to the VR-Forces API:

- Added ability to easily derive and use new *DtTerrainDatabase* classes.
- *DtNetIfModifyVrfObject* did not have myAppearance in the structure nor was it set over the network.
- *DtSymbolDrawer* in the *DtSymbol* object is now a pointer rather than a reference.
- The code that determines if an entity is considered to be under fire has been moved from *DtVrfObjectStateRepository* to damage actuators and the under fire detection sensor. The primary reason for this is that state repositories are designed to store state, not determine what the state should be. The determineUnderFire() member functions still have the same signatures and can be overridden from the damage actuators exactly as they could be from the *DtVrfObjectStateRepository*.
- The under fire parameters have been moved from *DtVrfObjectParameters* to *DtDamageComponentDescriptor*, however for backwards compatibility they can still be specified in their original location in the object parameters database. In that case they will override the values specified in the entry for the damage actuator.
- The DtSimObjectStateRepository::tick() member function has been removed because it was replaced by DtVrfObjectStateRepository::tick() in a previous version of VR-Forces. It has been removed as a clean-up measure; *DtVrfObject* ticks a *DtVrfObjectStateRepository*, not a *DtSimObjectStateRepository*.

- *T*he *DtSimpleStats* template class has been added. (The file *simpleStats.h* has been added to the geometry library.) It is for use in manual profiling. This is a simple class for maintaining statistics such as average, min, max, sum and count. The class intentionally does not provide support for more complex computations, nor does it maintain a list of N number of items added. This is to ensure that its performance is O(1) and its usage cost low enough for use in the most performance critical areas of code.
- *DtOutputLogger* is a simple class for logging information to a file. To be used, the caller must specify a logging level as well as a global logging level. If the threshold is less than the globalThreshold, then the output logger will output the specified information to the associated file, (as long as the file was created successfully). See the header file *outputLogger.h* for more information. It was added to the vrfUtil library.
- *DtVrfObjectManager* has a new member function, createSubObjectFromParameterEntry(). It provides a place for you to extend during the process of creating an aggregate with subordinates.
- In the class *DTVrfObjectParameters*, the member variables mySubordinateObjects and myAttachedObjects are now dynamically allocated, so you can override their creation and create derived types.
- In *DtGroundAutomotiveController* (*gndAutoCntlr.h*), the functions accelerationSoilFactor() and maxSpeedSoilFactor() are obsolete and will be removed in future versions of VR-Forces. They both return a value of 1.0, so neither incorporates effects due to soil types. The functions are being removed because the effects of soil factors on acceleration and speed are already taken into account in the *DtAutomotiveActuator*. By checking for these effects in both the controllers and the actuator, we were taking soil factors into account twice as much as we should have.
- To prevent the *DtWeaponController* from firing a pre-loaded munition at an entity that it does not have an ammoselect table entry for, the tick function now calls loadedAmmoValidForTarget(), which verifies that the ammo is loaded and appropriate for the target type before it calls fire().
- The myCurrentMunition and myCurrentFuze member variables were removed from *DtBaseWeaponController* because these values were moved to the *DtVrfWeaponPSR*. The values can now be accessed by calling weaponProcessState()->munitionLoader().currentMunition() or weaponProcessState()->munitionLoader().currentFuze().
- The class interface to *DtReaderWriter* has changed to allow an explicit search path to be used when reading and writing files. Previously, all relative filenames specified within a *DtReaderWriter* file were assigned to be relative to the application path. See *VR-Forces Developer's Guide* and *rdrWrtr.h* for details.

Ability to Check for VR-Forces Licenses at Runtime

VR-Forces can check the availability of VR-Forces licenses at runtime. This lets you write code to handle a situation in which an expected license is unavailable for some reason. For example, you might want to use the opportunity to pop up a warning to alert the user. You can check for the availablility of back-end, remote control API, or GUI licenses. For details, see *vrfCheckLicense.h*.

The following example shows how to embed a VR-Forces back-end in a manned simulator. Before trying to create an instance of the top-level back-end class (DtCgf), it checks to see if a back-end license is available. If not, it prints a warning message.

```
#include <vrfCheckLicense.h>

...

if (DtHaveVrfSimLicense() == true)
{
  DtCgf* cgf = new DtCgf(addr);
  cgf->init(exerciseConn);

  ...
}
else
{
  DtWarn << "Error checking out a VR-Forces back-end license."
    << std::endl;
  ...
}
```

getProcessSR() Member Functions Renamed

The getProcessSR() member functions were renamed to lookupAndCacheProcessSR() and the comments were improved to clarify that the member function is not an accessor (it returns void) and that it is used to update a class's pointer to its process state repository. In previous versions, some getProcessSR() member functions were public and some were protected. Now all lookupAndCacheProcessSR() are protected to indicate that this member function is to be used only to update these internal pointers. The affected classes are:

- *DtBallisticGunController*
- *DtTurretActuatorComponent*
- *DtSurfaceEntityPilotController*
- *DtRotaryWingPilotController*
- *DtPseudoAggregateSetController*
- *DtPseudoAggregateFormationController*
- *DtMissileControllerComponent*
- *DtIndirectFireWeaponController*
- *DtGroundAutoControllerComponent*
- *DtBaseWeaponController*

- *DtBallisticGunController*
- *DtFixedWingPilotController*.

DtCollisionDetector

Overall, the collision detector class was refactored to improve the interface, and make it much easier to override key aspects of the functionality. (For example, in order to change the behavior to return a list of obstructions, rather than the most imminent obstruction, or to institute a list of obstructions to ignore.) To this end, the following was done:

- The ability to remove object and vector obstructions IDs was added.
- A better, cleaner iterator type interface was added to protect against future changes in the API, as well as to make the interface clearer and easier to use.
- Typedefs have been added at the top of the class in order to further abstract the internal representation of the class away from the interface.
- The class now guarantees that only single instances of obstructions will be returned in the results. This motivated a change from using std::lists to std::sets, but that is an implementation detail. It is this type of change that the typedefs will help ameliorate.
- The list of obstructions are no longer cached internally. They are selected every collision query from the appropriate spatial manager.

The add and remove ID functions no longer take a container of IDs to add/remove, but rather a single ID. This is to remove the knowledge of what type of container the items are stored in the *DtObstructionSensor* from the *DtCollisionDetector*. That way, customer wishing to replace either will find it easier to do so.

Also, because the container types have changed as mentioned above, it is best to change any references to them in the code with typedef'd types at the top of the file. For example, in 3.7.1 API, you would have said the following:

```
std::set<DtVrfObject*> objectObstructions;
if (!collisionDetector.getObjectObstructions(objectObstructions))
{
  ...
}
```

Now, just say the following:

```
DtObjectObstructionContainer objectObstructions;
if (!collisionDetector.getObjectObstructions(objectObstructions))
{
  ...
}
```

DtObstructionSensor

The *DtObstructionSensor* has been updated to make it much easier to replace the collision detector that is uses to detect collisions. While not backwards compatible, this change makes it *much* easier to derive a new *DtCollisionDetector*, and specify that the *DtObstructionSensor* use it.

Specifically, functions for creating the collision detector, initializing the obstruction IDs to use from a descriptor and for calculating the stopping distance (part of the obstruction avoidance algorithm) also from a descriptor have been added.

See the header file for more details about the new functions:

```
virtual bool createCollisionDetector();
virtual bool initializeObstructionIDs(
  const DtObstructionSensorDescriptor& obstructionSensorDesc);
virtual bool calculateStoppingDistance(
  const DtObstructionSensorDescriptor& obstructionSensorDesc,
  double& stoppingDistance);
```

Setting Appearance Attributes

DtVrfRemoteController has a new member function, setAppearance(), that lets you set appearance attributes for a particular entity. A new message, *DtSetAppearanceRequest*, has been added to send this value to the back-end.

The member function processSetAppearance() in the *DtLocalEntitySetController* can be overridden in order to do special processing on appearance attribute values.

Task Manager/Task Controller Changes

DtPseudoAggregateTaskManager is obsolete. Aggregates should now be configured with the same *DtLocalTaskManager* class as individual entities.

Aggregate controller components are now responsible for sending clear task messages to the subordinates to which they assign tasks. This was formerly done by the *DtPseudoAggregateTaskManager* (now obsolete). This was moved to aggregate controllers because that is where the logic for issuing tasks to subordinates resides.

When you register for a task message in a *DtTaskController*, make sure you call startTask() before you set up any of the new task. The first thing startTask() does is clean out the old task in whatever controller was executing a top-level task at the time the new task came in (retask). This retask could involve the component who has received the new task, so you do not want to initialize anything before startTask() is invoked that will end up being cleaned out by clearTask().

Components derived from *DtTaskControllerComponent* will have their clearTask() member function called when either of the following is true:

- Skip task has been called, and task controller is currently executing a top-level task
- An entity has been retasked.

Change to *DtFltReader*

The xlateGroupChildrenIfAny() function now takes a reference to a pointer to a *DtGroup*, instead of an actual *DtGroup* pointer, so that it can allocate the group when and if a child node is translated. This avoids creation of *DtGroup* nodes that have no children.

Changes to *utils.h* and *utils.cxx*

The file I/O and general utility functions in *utils.h* have been updated to be compliant with our current coding specification. Additionally, several macros have been removed in favor of direct function calls that accomplish the same end. The following macros were removed:

- DEBUGMACRO
- PRINTINFO
- PRINTWARN

The VR-Link functions, *DtWarn*, *DtDebug*, and *DtInfo* should be used instead. Additionally, the function trap_to_debugger was removed. Instead, assert(), or some other suitable debugging function, should be used in its place.

Addition of *mathUtilities.h*

A new file of math related utility functions and classes has been added to the geometry project. This file will eventually be moved, but for now, it resides in the geometry project. The file contains math related utility functions, such as floating point comparison operations.

Deprecated Classes

Many areas of the code base have been deprecated that are already listed. Please see sections on the changes related to the *DtBoundingVolume* class for more information on functions deprecated as a result of the *DtBoundingVolume* refactoring.

DtSpatialIndex

The *DtSpatialIndex* class has been deprecated. It currently remains in the source tree as a test against the new *DtSpatialSubdivision* framework (only for terrain spatial organization though). The *DtTerrainDatabase* no longer contains a *DtSpatialIndex*, and instead now contains a *DtSpatialSubdivision*. The life cycle functions (constructors, and so on) of the *DtSpatialSubdivision* are the same or extremely similar to the *DtSpatialIndex*. As a result, changing code that created a *DtSpatialIndex* to create a *DtSpatialSubdivision* is the following:

Existing method

```
DtSpatialIndex* spatialIndex = new DtSpatialIndex(params...);
```

New method:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision =
  new DtSpatialSubdivision<DtGdbNode*>(params...);
```

The spatial subdivision is a templated container. As such the largest difference between creating/using the spatial index and the spatial subdivision is the need to specify the template parameter. For more information about templates, please consult a C++ reference book.

Changes to *dyad.h* and *dyad.cxx*

The code has been updated to account for new changes in the use of bounding volumes (and intersecting them). Additionally, the code in the *dyad.h* and *dyad.cxx* files was refactored to improve understandability as well as correctness. Mostly, all members (with very few exceptions) have been renamed from _"varname" to myVarName. As many of these are public or exposed to friend functions, this helps tremendously when dealing with the classes as it is clear what is member data and what is not.

Changes to *intersection.h*

The functions in *intersection.h* were removed, except for GtIntersectionUniqueSolution(), which was moved into new *mathUtilities.h* and *mathUtilities.cxx* files in the geometry project. The functions were not used anywhere in the code base and were outdated.

Additionally, GtIntersectionUniqueSolution() was renamed DtIntersectionUniqueSolution() to be consistent with the source code, and the last two parameters were removed from the function as they were not used. To update existing references to this function, simply remove the last two parameters (and rename the function).

Addition of *geometricUtilities.h*

A new file utility functions for geometric calculations has been added to the geometry project (*geometry.lib*). (This file may eventually be moved.) The file contains functions to calculate an extent from a bounding volume - either by the bounding volume's radius, or by the actual transformed bounding volume (bounding volume in the reference coordinate system). For more information on how this calculation is performed, see the bounding volume header file. These functions are used by objects to create an extent from their bounding volumes, for use in intersecting with a spatial subdivision.

New Examples

VR-Forces has the following new examples:

- envProcessListen demonstrates how to listen for and display state updates for environmental process objects (control and overlay objects) generated by VR-Forces.
- addAttr demonstrates how to extend the RPR FOM. It is derived from the VR-Link addAttr example.
- chat demonstrates how to derive a GUI remote control class and install it.
- interfaceContent - example of sending a new type of *DtSimInterfaceContent* subclass which contains a string and number from the front-end to the back-end and vice versa.

Documentation Updates

The print and PDF versions of all VR-Forces manuals have been updated for this release.

Bug Fixes

This release fixes the following problems:

- VR-Forces no longer crashes if you click on empty space in the Resources tab of the Entity Information dialog box.
- VR-Forces now correctly displays velocity and acceleration in UTM databases in the Entity Information dialog box.
- VR-Forces no longer crashes if you run a logger recording with an incorrect database loaded or no database opened.
- When selecting a feature type that might be both an areal and point feature, make sure to select both kinds.
- Ground entities no longer continue to move along the path of a route that is deleted while the entity is moving along it.
- Shape files containing NULL records are now being imported properly.
- Fixed an incomplete #ifdef in backwards compatibility header file *vrfObjBaseSR.h*.

- There was a problem with calculation of the intersection record normal when the terrain contained coordinate system nodes. The correct normal will now be returned if requested.
- Subsurface entities will now work with terrain databases that have both underwater terrain polygons and water surface polygons. You can enable terrain checking to perform terrain avoidance (by setting the terrain-check parameter to True), and the entity will no longer crash when set to an altitude below the surface.
- Fixed the move to altitude task for fixed wing and rotary wing entities so that its state gets checkpointed into the respective process state repository.
- Fixed the createWorkingControlPoint() member functions in the fixed and rotary wing move to controllers so they properly clean up the waypoint in case the create() or init() methods called on it fail.
- The state repository tick rate parameters are now implemented. See "Tuning the State Repository Tick Rate," on page 12, in *VR-Forces Configuration Guide*.
- DtVrfSubObjectParameterList::getData() now uses the *DtReaderWriter* name read in for a list element, instead of hard-coding it to the string sub-object.
- In plans, executing conditional statements that only contain set data requests no longer causes the entity to skip to the next task in the plan if it has not finished the current task.
- The **max-elevation-range** and **min-elevation-range** parameters of the **ballistic-gun-descriptor** were being specified in degrees. They have now been updated to use radians.
- The list of subordinate entities and subaggregrates in remote aggregates' *DtVrfSubordinateManager* are now kept up to date.
- The Fire Cruise Missile dialog allows missile speed entry of > 1000 m/s.
- The altitude published for IFF mode 5 now works with geocentric databases as well as projected databases. Previously it only worked with projected databases.
- Subsurface entities will now work with terrain databases that have both underwater terrain polygons and water surface polygons. You can enable terrain checking to perform terrain avoidance (by setting the terrain-check parameter to True), and the entity will no longer crash when set to an altitude below the surface.
- Fixed-wing and rotary-wing entities now get their state properly checkpointed when conducting a move to altitude task.
- Fixed the createWorkingControlPoint() member functions in the fixed and rotary wing move to controllers so they properly clean up the waypoint in case the create() or init() methods called on it fail.
- Ground vehicles no longer are able to exceed their maximum speed, except when they are traveling downhill.
- The Ocean polygons in the Berkeley terrain have been fixed to no longer have the grass soil type.
- DFAD files are now imported correctly.

- Several bugs in creating surface characteristics from texture file names, using the *surfChar.map* file, when importing OpenFlight files have been fixed. The surface characteristic for Sand is now properly created when a texture name contains the string "sand". In general, creating surface characteristics from the texture names and the surface characteristics map has been improved.
- Specifying a larger minimum than maximum longitude or latitude in DtedMetafileRecords no longer causes an application crash. If the minimum is larger than the maximum, the specified values are ignored and a warning is output.
- The CTDB file importer now treats linear features with only a single vertex as point features representing trees. Previously these features were ignored, which was incorrect.
- OpenFlight files that contained relative pathnames to referenced files, textures, and so on, now import correctly.
- DtGdbNodes marked for removal are now not written out to disk in the GDB file. This results in a small improvement in memory usage and performance.
- The UTM Zone specified in the OpenFlight header is now respected. Previously, the UTM zone was calculated from the specified origin latitude/longitude. (Users seeing apparent projection errors when importing OpenFlight files created by MPI's CTS product should see an improvement.)
- A memory leak in the vector obstruction code has been fixed. The memory leak only occurred under Linux, and only when avoiding vector obstructions.
- Civilian vehicle *.ope* files are now correctly configured for obstruction avoidance. Previously, the entry for the collision avoidance controller was after the move-to-path controller entry, which was incorrect.
- Individual lifeforms now respect the surface soil types when moving. This was broken in the previous release, but is now fixed.
- Vector obstructions are now properly created from the original vector features. In VR-Forces 3.7.1, the length and width components of the features were being switched unintentionally. This is now fixed. (In general, the entire vector obstruction creation has been simplified and corrected.)
- The map datum for Open Flight files was being read and set incorrectly. This has been fixed. If you have Open Flight files with a non-WGS-84 map datum, you should reimport them into the GDB format.
- The rotary-wing state repository now ignores water surfaces when placing subsurface entities. This a bug that clamped subsurface entities to the water polygons as ground. They can now be set to be underwater. This does not affect initial placement and creation.
- The map datum was being read and set incorrectly

Known Problems

VR-Forces Release 3.8 has the following known problems:

- If you are using VR-Forces in a DIS exercise, you may experience problems with dropped packets and entities may time out. To avoid this problem, you can use asynchronous I/O by starting with the `-I` parameter.
- Aggregates composed of fixed-wing entities can only perform limited behaviors as an aggregate.
- 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.
- Any while loop that evaluates to TRUE that has only set data requests in the body of the condition crashes the back-end.
- Fixed-wing take-off, landing, and taxiing behaviors may be difficult to configure through the front-end when simulating on geocentric terrain databases. This is due to the fact that it uses a different projected terrain database in the front-end when simulating on a geocentric terrain database in the back-end. Correlation errors in terrain elevation between the two databases must be small (on the order of meters) when placing waypoints or routes on the ground to support fixed-wing ground behaviors. Either use high resolution terrain databases (minimizing the elevation error between the front-end and back-end databases) when configuring these behaviors, or restrict your scenarios to fixed-wing flying behaviors.
- Using the *tdbtool* 's balancing option (`-b br,lf`) to balance certain terrains may cause the generated *.gdb* file to display incorrectly. This has to do with a draw ordering bug that is not currently corrected for all terrains. The resulting GDB file will still give correct intersection information, so it will still be suitable for simulation purposes.
- Vector clipping does not work properly with areal features.
- Surface entities are not configured to fire weapons. You can customize the object parameters database entries to enable this.
- The Qt translation files (in *./translations*) for built-in Qt-level GUI items do not work properly. The translation files for VR-Forces work correctly.
- If you are running VR-Forces on Linux with the DMSO RTI, you cannot run in combined mode. Start the GUI and back-end separately.

- The order in which you create plans for aggregate units and individual members of aggregate units affects how the plans are implemented. If you create a plan for an aggregate member and then create a plan for the aggregate, the aggregate plan overrides the individual plan. If you create a plan for an aggregate and then create a plan for a member of the aggregate, the member plan is executed.
- If you run a MÄK Logger recording of a VR-Forces simulation and the site:application number for the back-end is the same as the site:application number of the back-end used to create the recording, VR-Forces may crash. To avoid this problem make sure that you run VR-Forces with different site:application numbers, or just run the front-end and use it as a plan view display to view the recording.
- The component descriptions in Appendix C of *VR-Forces Developer's Guide* have not been updated to reflect changes made in recent releases of VR-Forces. In particular references to port interfaces are obsolete and the component descriptions do not reflect the use of process state repositories. Fixed-wing and rotary-wing components may be significantly incorrect.
- Using large filled polygons, such as minefields, in overlays can degrade performance.
- You cannot load vector data with geocentric databases. Vector data works only with geodetic or UTM databases.
- You cannot give a helicopter a Set Altitude request that places it at less than the starting height value.
- Plans stop executing if they encounter a task that they cannot execute. For example, if the plan for a rotary-wing entity include Turn To Heading, which is not a valid task for rotary-wing entities, the plan stops executing.
- If you delete an entity that has a plan and create a new entity with the same name as the deleted entity, the old plan is displayed in the new entity's Edit Plan window. However, the new entity cannot execute that plan.
- Missiles get initialized at their maximum velocity. They do not accelerate to maximum velocity.
- Torpedoes ignore a polygon with soil type water, but the intervisibility fan does not. So it does not show an accurate representation of who can see who.
- When a subsurface entity gets destroyed it jumps to altitude 0.
- It you task a rotary-wing entity to fly to a waypoint on the ground, it will land when it gets to the waypoint. If opposing force trucks get within range of the rotary wing it will fire. Since the rotary-wing entity is at a slight incline, it will fire directly into the ground and explode.
- Intervisibility works with linear and areal features, but not with point features.
- Occasionally, VR-Forces is unable to load a scenario. When this happens, delete the contents of the *./data/temp* directory and try loading again.
- Setting the heading of a rotary wing entity whose altitude is less than the **starting-height** parameter causes it to rise up to the **starting-height** value. The workaround is to set a very small (non-zero) **starting-height**.

- If a task is interrupted by a trigger that only has set statements, after the set statements are executed, the plan moves to the next task (if there is one) rather than continuing with the task that was interrupted, as it is supposed to. If there is no additional task in the plan, the entity resumes executing the interrupted task.
- You cannot use translation files to translate entity labels.
- You cannot use translation files to translate labels in the Create Entity menu (vrf.ent).