VR-inTerra

Developers Guide

VT MÄK
A company of VT Systems

VR-inTerra

Developers Guide

VT MÄK
A company of VT Systems

For third-party license information, please see “Third Party Licenses,” on page xx.

VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

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

info@mak.com

www.mak.com

Revision VRI-1.0-1-110805

Contents

Chapter 3 The Terrain API

Chapter 4 The MÄK Specification Model

MÄK Products Glossary

Index

Preface

This manual is written for persons who will develop software using the VR-inTerra SDK. The manual assumes that you are familiar with your operating system, development environment, and C++.

Please see release documentation for updates to this manual and the latest information about your version of VR-inTerra.

How This Manual is Organized

The chapters are organized as follows:

Chapter 1, *Introduction* introduces you to the VR-inTerra toolkit.

Chapter 2, *Installing VR-inTerra* explains how to install VR-inTerra and set up license management.

Chapter 3, *The Terrain API* describes the VR-inTerra API.

Chapter 4, *The MÄK Specification Model* descries the MÄK Specification Model for classifying feature data.

MÄK Products Glossary is a glossary of terms common to MÄK products and the modeling and simulation industry.

VR-inTerra Documentation

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

- *VR-inTerra Developers Guide* explains how to use the VR-inTerral API.
- Class documentation. Classes are documented in linked HTML pages.
- *VR-inTerra Release Notes*.

MÄK Products

VR-inTerra is a member of the VT MÄK line of software products designed to streamline the process of developing and using networked simulated environments. The VT MÄK product line includes the following:

- VR-Link® Network Toolkit. VR-Link is an object-oriented library of C++ functions and definitions that implement the High Level Architecture (HLA) and the Distributed Interactive Simulation (DIS) protocol. VR-Link has built-in support for the RPR FOM and allows you to map to other FOMs. This library minimizes the time and effort required to build and maintain new HLA or DIS-compliant applications, and to integrate such compliance into existing applications.

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

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

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

- VR-Vantage™. VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes four end-user applications (VR-Vantage Stealth, VR-Vantage XR, VR-Vantage PVD, and VR-Vantage IG), the VR-Vantage Toolkit, and VR-Vantage FreeView.
  - VR-Vantage Stealth displays a realistic, 3D view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway.
  - VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.

  - VR-Vantage XR provides a common operational picture using the same 3D views as VR-Vantage Stealth, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world.
  - VR-Vantage PVD provides a 2D plan view display. It gives you the big picture of the simulated world.
  - The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.
  - VR-Vantage Free View is a terrain and 3D model viewer that introduces some of the features of VR-Vantage applications in a useful, free utility.
- MÄK Data Logger. The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.
- VR-inTerra. VR-inTerra is a C++ API for adding terrain agility to applications. It can load, page, or stream terrain from a wide variety of formats or sources into a single, consistent run-time representation consisting of a collision graph and vector network.

How to Contact Us

For VR-inTerra technical support, information about upgrades, and information about other MÄK products, you can contact us in the following ways:

Telephone

Call or fax us at:	Voice:	617-876-8085 (extension 3 for support)
	Fax:	617-876-9208

E-mail

Sales and upgrade information:	info@mak.com
Technical support:	support@mak.com
VR-Vantage support:	vrv-support@mak.com

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/licenses.php
Product version and platform information:	www.mak.com/support/productversions.php
For the free, unlicensed MÄK RTI:	www.mak.com/products/rti.php
Support FAQ:	www.mak.com/support/faq.php

Post

Send postal correspondence to:	VT MÄK 68 Moulton St. Cambridge, MA, USA 02138

When requesting support, please tell us the product you are using, the version, and the platform on which you are running.

Document Conventions

This manual uses the following typographic conventions:

`Monospaced`	Indicates commands or values you enter.
`Monospaced Bold`	Indicates a key on the keyboard.
`Monospaced Italic`	Indicates command variables that you replace with appropriate values.
`{ }`	Indicates required arguments.
`[ ]`	Indicates optional arguments.
|	Separates options in a command where only one option may be chosen at a time.
`(	)`	In command syntax, indicates equivalent alternatives for a command-line option, for example, `(-h | --help)`.
/	Indicates a directory. Since MÄK products run on both UNIX and Windows PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Italic	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose **File → Save**.
	Indicates supplemental or clarifying information.
	Indicates additional information that you must observe to ensure the success of a procedure or other task.

Directory names are preceded with dot and slash characters that show their position with respect to the VR-inTerra home directory. For example, the directory *vrforces/doc* appears in the text as *./doc*.

Hardware and Operating System Naming Conventions

Unless otherwise specified, references to UNIX include Linux, regardless of the type of computer it is running on. References to PCs refer to Intel processor-compatible computers running a version of Microsoft Windows.

Mouse Button Naming Conventions

An instruction to click the mouse button, refers to clicking the primary mouse button, usually the left button for right-handed mice and the right button for left-handed mice. The popup menu refers to the menu displayed when you click the secondary mouse button, usually the right button on right-handed mice and the left button on left-handed mice.

Third Party Licenses

MÄK software products may use code from third parties. This section contains the license documentation required by these third parties.

Boost License

VR-Link, and all MÄK software that uses VR-Link uses some code which is distributed under the Boost License. All header files that contain Boost code are properly attributed. The boost web site is: www.boost.org.

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

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

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

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

libXML and libICONV

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

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

pThreads Library

VR-Exchange links with the pThreads win32 library. The library is distributed under the GNU Lesser General Public License (LGPL). MÄK has made no modification to this library. For information about the pThreads win32 library please see: http://sourceware.org/pthreads-win32/

EHS, PCRE, and PME

The MÄK RTI links with the EHS, PCRE, and PME libraries. These libraries are distributed under the GNU Lesser General Public License (LGPL). MÄK has made modification to these libraries only to allow them to compile on the supported platforms. For more information about these libraries please see the following web sites:

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

Freefont OpenType Font Set

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

Introduction

This chapter provides a general introduction to VR-inTerra.

1.1. Introduction to VR-inTerra

VR-inTerra addresses the terrain interoperability problems that occur when multiple simulation and visualization applications operate in distributed simulation environments. It brings Terrain Agility to simulation applications.

Terrain interoperability problems stem from two major challenges facing simulation applications users when determining their terrain needs - acquiring terrain that has the required resolution and scope (coverage area) and acquiring terrain that their applications can load and comprehend. Terrain Agility allows new simulations to use a wide range of terrain formats that are already in use by existing simulation and visualization systems, thereby extending the life and utility of legacy simulations.

VR-inTerra is a C++ API that enables simulation applications to natively take advantage of new terrain approaches like dynamic composition, direct-from-source terrain, and open streaming terrain. Due to Terrain Agility, simulation applications are able to play in synthetic environments without having to support the myriad terrain formats and to convert other terrain to their own proprietary formats.

VR-inTerra takes care of loading or streaming the terrain and provides the simulation with access to a collision graph and a feature graph. VR-inTerra's collision graph allows for efficient height above terrain and line of sight intersections at extremely high rates which are required by today's increasingly complex simulators. The feature graph provides the data needed for navigation and route planning along, and through, this terrain.

Using VR-inTerra, a simulation can be terrain agile and can achieve high degrees of correlation with other terrain agile visual and simulation applications, as well as command and control systems that use the same geospatial formats.

Installing VR-inTerra

This chapter explains how to install VR-inTerra and set up license management.

2.1. Installing VR-inTerra

This section explains how to install VR-inTerra on a Windows or a Linux operating system. You must also install the license manager files. (For details, please see "Installing and Setting Up the License Manager," on page 2-5.)

2.1.1. Installing VR-inTerra on Windows

Before you install VR-inTerra, please read *VR-inTerra Release Notes* to see if there are any special instructions for installation.

Installing VR-inTerra from Optical Media

MÄK products are supplied on CDs or DVDs with a common installation interface.

To install VR-inTerra:

1. Insert the product disc into your optical media drive. If the autorun feature on your computer is enabled, the MÄK PC Products Installation window opens. The PC Products Installation window is similar to Figure 2-1.

   If the MÄK PC Products Installation window does not open, open the Windows Explorer. In the root directory for your optical media drive, double-click *makInst.exe*. The MÄK PC Products Installation window opens.

2. In the MÄK PC Products Installation window, click VR-inTerra.

3. Follow the instructions of the installation program.

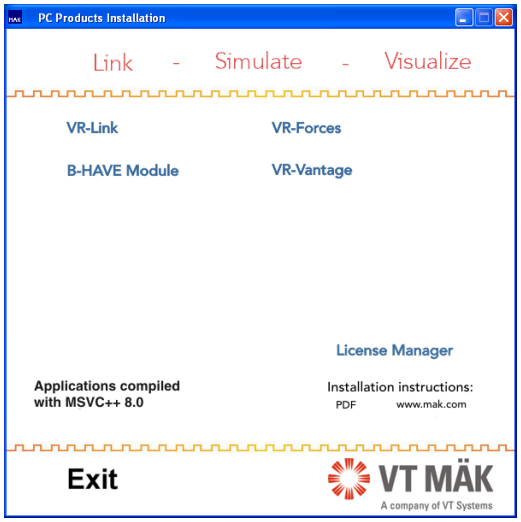

Figure 2-1. MÄK PC Products Installation window

Installing VR-inTerra from a Downloaded File

If you downloaded VR-inTerra, you received .

➤ To install VR-inTerra from an executable file, run the executable. Follow the installation instructions in "Installing VR-inTerra from Optical Media" starting with step 3.

2.1.2. Installing VR-inTerra on a Linux System

Before you install VR-inTerra, please read *VR-inTerra Release Notes* to see if there are any special instructions for installation.

Installing VR-inTerra from Optical Media

To install VR-inTerra:

1. Mount the optical drive.
2. Insert the MÄK Products disc into your optical media drive.
3. Run:

   ```
   ./install
   ```

4. Follow the on-screen instructions. You can install VR-inTerra into any directory for which you have write permission.

Installing VR-inTerra from a Downloaded File

If you downloaded VR-inTerra, you received a compressed tar file.

To install VR-inTerra from a tar file:

1. Create the directory in which you want to install VR-inTerra.
2. Copy the tar file to the install directory.
3. Uncompress and untar the tar file:

   `gtar xzf stealth`*`xx-os`*`.tar.gz`

   where *`xx-os`* is the release, operating system, and other product release coding.

! You must use gtar (from GNU) not the operating system's default tar.

2.2. The VR-inTerra Directory Structure

The installation process creates the VR-inTerra directory tree shown in Figure 2-2 (runtime and toolkit). Table 2-1 describe the contents of the directories.

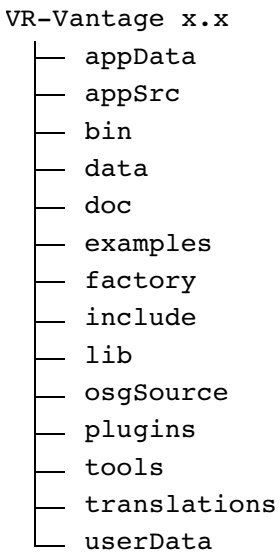

Figure 2-2. VR-inTerra directory tree

Table 2-1 describes the contents of the directories.

Table 2-1: Contents of VR-inTerra directories

Directory	Directory contents
./appData	Subdirectories contain data required by the VR-inTerra applications, such as model definitions and saved settings.
./appSrc	Source code for rebuilding the VR-inTerra applications.
./bin	Executables and dynamic link libraries.
./data	Sample databases, images, and vehicle models.
./doc	VR-inTerra documentation and class reference files.
./examples	Examples for the VR-inTerra Toolkit API.
./factory	Default settings (for reverting to “factory” settings).
./include	Header files for the VR-inTerra Toolkit API.
./lib	Libraries for building applications with the VR-inTerra Toolkit API.
./osgSource	The OSG and third party source used to build VR-inTerra.
./plugins	Plug-in shared libraries.

Table 2-1: Contents of VR-inTerra directories

Directory	Directory contents
./tools	Helper applications such as MÄK Terrain Database Tool and the CIGI Host Emulator.
./translations	Translation files for localizing VR-inTerra menus.
./userData	Terrain and scene files saved by application users.

2.3. Installing and Setting Up the License Manager

Before you can use a MÄK product, you must obtain a valid license file, install the License Manager, and configure the license server and client machines. The License Manager uses a client-server architecture, so you do not need to install the License Manager on every computer on which you install MÄK products. You only need to install it on the computer that you will use as the license server.

If you have already installed the License Manager for another MÄK product, you do not have to install it again. You just need to make sure you have licenses for your newly installed products.

The License Manager installer is included on MÄK installation media. It is separate from the product installers. The installers are also available for download at:

- Windows: ftp://ftp.mak.com/out/MAKLicenseManager-win-setup.exe
- Linux: ftp://ftp.mak.com/out/MAKLicenseManager-linux-setup.tar.gz

Complete installation and configuration instructions are included with the license manager installer. Instructions are also available at: http://www.mak.com/products/install_license.php

Some customers use dongle licenses instead of running a license server. Instructions for using dongles are in the license manager documentation.

2.3.1. Specifying the License Server

The first time you run a MÄK application on a particular computer, the License Setup dialog box opens (Figure 2-3). It prompts you to enter the hostname of the license server and optionally, a port number.

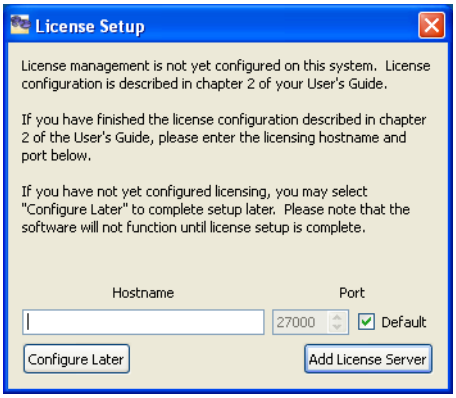

Figure 2-3. License Setup dialog box

If you do not know the hostname of the license server, click Configure Later. When you have the hostname, you can start the application again and complete the dialog box. You will not be able to run any MÄK applications until you set up license management.

If you know the hostname, type it in the Hostname box. Then click Add License Server. The application will start.

- If you are running MÄK products on the license server machine, it is also a client, so you must specify the license server on that machine too.
- If you change the license server, the saved configuration will no longer be valid and the License Setup dialog box will open the next time you start a MÄK application.
- You can clear the saved license configuration by deleting the cache file. On Windows it is *C:\Documents and Settings\user_name\Application Data\MAK\licenses<n>.xml*. On UNIX, it is *.mak/licenses<n>.xml*. (There may be more than one cache file, for example, *licenses1.xml* and *licenses2.xml*.)

The MAKLMGRD_LICENSE_FILE Environment Variable

As an alternative to using the License Setup procedure described in the previous section, you can configure the license server in an environment variable. The MAKLMGRD_LICENSE_FILE environment variable identifies the server machine. If you set this environment variable, it overrides the settings stored by the License Setup procedure.

To view a video that illustrates this procedure on Windows, please go to http://www.mak.com/html/setenvvar.htm

The syntax for the environment variable is: `@Server_name`. For example, if the server machine is oak, set the environment variable to `@oak`.

The following sections explain how to set environment variables on the different platforms that MÄK products run on.

Windows

To add the MAKLMGRD_LICENSE_FILE in Windows:

1. Open the System Properties dialog box.

   Windows XP:

   a. In Windows XP, on the Start menu, choose My Computer.

   b. In the My Computer window, under System Tasks, click View System Information. The System Properties dialog box opens.

   Windows Vista or Windows 7:

   a. In Windows Vista, open the Control Panel. On Vista, select System and Maintenance.

   b. Click System.

   c. Click Advanced System Settings. The System Properties dialog box opens.

2. Click the Advanced tab.
3. Click the Environment Variables button. The Environment Variables dialog box opens.
4. Click the New button. The New System Variable dialog box opens.
5. In the Variable Name field, enter MAKLMGRD_LICENSE_FILE.
6. In the Variable Value field, enter `@server_name`, where `server_name` is the name of the license server.
7. Click OK to back out of each dialog box and set the variable.

Linux

On Linux, you set environment variables in your *.cshrc* (or equivalent startup file). Set the variable similarly to the following example:

```
setenv MAKLMGRD_LICENSE_FILE @oak
```

If you are using the sh or bash shells, you set environment variables in your *.profile* file (or *.bashrc*). Set the variable similarly to the following example:

```
MAKLMGRD_LICENSE_FILE=@oak
export MAKLMGRD_LICENSE_FILE
```

Do not put spaces around the equal (=) sign.

You are ready to run the license server and use your new licenses or MÄK products.

The Terrain API

This chapter describes how to use the Terrain API, including how to create terrain databases and work with vector data.

3.1. Introduction

VR-inTerra is a set of classes that enable applications to read, write, and query terrain databases. Using VR-inTerra you can:

- Load terrain databases in formats such as MÄK GDB, OpenFlight, ESRI Shapefile, and CTDB.
- Perform intersection tests with the terrain geometry and vector network
- Query the vector network for information about feature data.

3.2. Using the DtTerrainInterface Class

The *DtTerrainInterface* class (*terrainInterface.h*) contains the representation of terrain skin and terrain features. The terrain skin consists of the geometry of the terrain (typically represented as polygons) and its surface characteristics (that is, soil type). Terrain features include natural and man-made items such as trees, rivers, roads, buildings, and so on. Terrain features are represented as a vector network, comprised of points, lines, and areas.

The *DtTerrainInterface* class uses a pointer to a *DtTerrainImplementation* to manage the specifics of how each terrain type operates. The *DtTerrainInterface* maintains a factory of implementation types and chooses the correct type during the terrain loading process. Additionally *DtTerrainPassThroughLayers* are also terrain implementations, but they pass the calls to an underlying *DtTerrainImplementation*. Any number of pass through layers can be added on to of a loaded terrain to modify the terrain implementations.

Additionally after a terrain is loaded the *DtTerrainCapabilities* can be accessed from the terrain interface. This provide information about the terrain that was loaded.

3.2.1. Terrain Intersection Tests

The *DtTerrainInterface* class has an interface that lets you query for information about the terrain data. One of the most common queries made to the terrain database is for a terrain intersection. Terrain intersections can be used in ground entity models to determine how to position and orient entities on the terrain surface.

The following example shows how to query the terrain database for an intersection with the terrain surface, returning the location of the intersection only:

```
// Set up chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

DtPoint isectPoint;
double intersectionTime;

// Calculate intersection with the terrain skin
if(terrainInterface->intersect(chord, isectPoint, intersectionTime) ==
  true)
{
  DtInfo("Intersected terrain at location %s\n", isectPoint.string());
}
```

You can make more complex queries to the terrain interface, requesting such information as the list of all intersections a given chord makes with the terrain, surface type information at the point of intersection, and intersections with features in the vector network. For more details about terrain intersections, please see "Querying the Terrain Interface," on page 3-6.

3.2.2. Coordinate Systems

A *DtTerrainInterface* has a coordinate system, which defines the origin and type of coordinates being used in the database that was loaded. The *Coordinate_System* class (*coordSystem.h*) represents the coordinate system for the database. The *Coordinate_System* class provides methods for transforming a coordinate between the local coordinate system (the coordinate system used in the database) and the geocentric coordinate system. The derived types of *Coordinate_System* classes include:

- *DtGeodeticCoordinateSystem* – Geodetic coordinates (latitude, longitude, altitude)
- *UTM_Coordinate_System* – Universal Transverse Mercator
- *lambertConformalProjection* – Lambert Conformal Conical.

The *DtTerrainInterface* class has methods for setting and retrieving the coordinate system used in the database. It also has methods for transforming from its current coordinate system to any other coordinate system, performing the necessary transformations on loaded terrain data as appropriate.

3.2.3. Adding a *DtTerrainPassThroughLayer*

Terrain pass through layers can be added to the stack of terrain implementations by calling addTerrainPassThroughLayer() on the *DtTerrainInterface* class. This will instantiate a new pass through layer on the top of the terrain implementation stack. By doing this you now can intercept terrain calls as they are being made to the terrain implementation and modify them if the need arises.

An example pass through is the *DtIntersectTimerPassThroughLayer* class. This class times the intersect calls as they are being made and reports the average times when asked.

The following code adds the *DtIntersectTimerPassThrough*.

```
DtTerrainInterface terrainInterface;
terrainInterface.init();
terrainInterface.loadTerrain("someTerrain");
terrainInterface.addTerrainPassThroughLayer("timer");
```

3.2.4. Terrain Surface (Soil Type)

When querying for an intersection with the terrain a chord terrain intersection record can be returned. One of the important aspects of this record is the *DtSurface*, which holds the characteristics of a surface, including its color and the nature of the surface. A *DtSurface* has the following attributes:

- DtColor – RGB color.
- DtMedium – for example, ground, water.
- DtEnvState – environmental state, for example, dry, icy, wet.
- DtSoilType – for example, paved road, swamp, muddy, and so on.
- DtRoughnessSoilType – derived from soil type and used in modeling ground vehicle behavior (moving factor efficiency).

Color is an RGB value associated with the surface. This color is used when drawing the terrain database in a graphical display.

The medium, environmental state, and soil type attributes are represented by a set of enumerated types defined in the *DtSurface* class. Please see *surface.h* for the complete set of enumerations.

The DtRoughnessSoilType is a classification of terrain surface that is suitable for use in simulators that need to model the effects of soil type on ground vehicle behavior. It is derived from the soil type parameter. (Soil types are automatically mapped to roughness soil types in the *DtSurface* class.)

DtSurfaces are stored in a *DtSurfaceManager*, a member of the *DtTerrainInterface*. Individual triangles and polygons just hold a reference to a *DtSurface* in the *DtSurfaceManager*.

3.2.5. Terrain Capabilities

After loading a terrain into the terrain interface the *DtTerrainCapabilities* class is filled out and available from the DtTerrainInterface::TerrainCapabilities() method. The capabilities class contains a list of key value pairs that describe the pertinent information about the terrain.

The most important item that is found in the terrain capabilities class is whether the terrain supports preloading. This is useful with paged terrains and is discussed later in "Synchronous and Asynchronous Calls with Paged Terrains," on page 3-14.

3.2.6. Terrain File Locations

One of the import aspects of the terrain interface is to set up the file locations that the interface will use when locating configuration files and terrain data. The *DtTerrainFileLocations* class can be used before the creation of the *DtTerrainInterface* class to change the default locations of these files. The terrain subsystem is interested in the following directories:

- *../userData*. The *userData* directory contains data that is expected to be written to by users, including server and terrain configuration files.
- *../appData*. The *appData* directory contains data that is expected to not be changed by users and contains the ModelDefinitions files.
- *../data*. The *data* directory usually contains the bulk data such as terrain polygon data and models.
- *../plugins*. The *plugins* directory contains plug-ins to the VR-inTerra runtime. The VR-Vantage terrain implementation is one such plug-in.

All of these directories can be referenced in file locations by using $(DATA_DIR), $(USER_DIR), $(APP_DIR), and $(PLUGINS_DIR) in file names and data files. For example you can load a terrain by specifying *$(DATA_DIR)\terrains\Makland.mtf*, instead of having to specify the path directly.

The *DtTerrainFileLocations* class can be used to make the substitutions using the replaceString() or replaceAll() methods.

3.3. Querying the Terrain Interface

DtTerrainInterface provides a set of intersection methods you can use to query the terrain database, using 3D chords to define the segment and direction of interest for intersections.

You can ask:

- Is there an intersection?
- What is the first point at which an object intersects the terrain?
- What are all of the intersections along the chord?
- What is the closest intersection, along the down/up vectors to the specified location?
- What is the height of the terrain at a specified location?

Additionally, for a given intersection query you can specify what kinds of information you want to receive:

- Boolean: indicating intersection or no intersection (no additional data)
- Point: coordinates of the intersection point
- Normal: the normal vector in the intersection point
- Vector network intersections: first (best) intersection, or list of intersections along the chord.

For example, to query the terrain interface for the intersection with the terrain surface, returning the location of the intersection only, do the following:

```
// Set up chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

DtPoint isectPoint;
double intersectionTime;

// Calculate intersection with the terrain skin
if(terrainInterface->intersect(chord, isectPoint, intersectionTime))
{
  DtInfo("Intersected terrain at location %s\n", isectPoint.string());
}
```

You can use the **intersectionTime** parameter returned in the intersect() method to determine where along the chord the intersection with the terrain occurred. Values for intersection time *t*, given a chord(*p1*, *p2*) indicate the following:

- $t < 0$ – intersection lies on an extension of the chord before *p1*
- $0 <= t <= 1$ – intersection lies on the chord
- $t > 1$ – intersection lies on an extension of chord after *p2*.

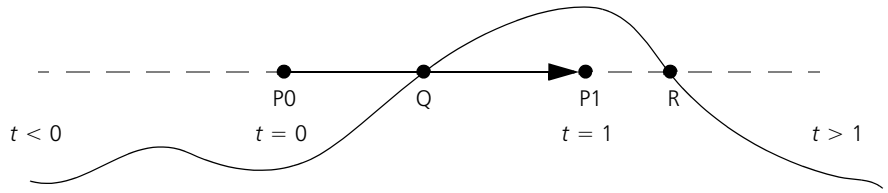

Figure 3-1. Intersection point

In the Figure 3-1, a call to intersect() with chord (P0, P1) intersects the terrain at point Q. The method will return TRUE, because point Q lies along the chord, and the intersection time at Q will be a value between 0 and 1. The intersection time at point R (which lies on an extension of the chord) has an intersection time > 1.

To get more information about the intersection point, you can call a variant of intersect() that returns a *DtChordIntersectionRecord* (*chrdIntRec.h*), instead of just the *DtPoint*. The *DtChordIntersectionRecord* can return information such as the location of the intersection, surface type, normal vector of the terrain at that point, and the polygon intersected. For example, to make an intersection test (similar to the previous example), requesting the intersection point and normal, do the following:

```
// Set up chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

// Return intersection details in intersectRecord
DtChordIntersectionRecord intersectRecord;

// Calculate intersection with terrain skin, requesting that the normal
// and the location of the intersection point be retrieved.
DtGdbNodeDtIntersectRecordType flag =
  (DtGdbNodeDtIntersectRecordType)(DtGdbNodeIRT_IPOINT |
  DtGdbNodeIRT_INORMAL);

// Try to intersect the terrain to get the left triangle
if(terrainInterface.intersect(chord, intersectionRecord, flag))
{
  DtInfo("Normal at terrain intersection %s is %s\n",
    intersectRecord.intesectionPoint().string(),
    intersectRecord.normal().string());
}
```

3.4. Using Vector Networks

A vector network is a structure of vectors and points used to represent features in the database such as rivers, roads, and buildings. A *DtTerrainInterface* represents the vector network in the following nodes:

- *DtVectorNetwork* (*vecNetwrk.h*): holds the actual set of vectors, points, and edges
- *DtSpecificationManager* (*specMgr.h*): a member of the *DtVectorNetwork* that manages the specifications (collections of attributes describing a feature) associated with items in the vector network.

Features are represented in the vector network using the following nodes:

- *DtNetworkNode* (*netwrkNode.h*): Point features (such as a tree or building), or junctions between edges.
- *DtNetworkEdge* (*netwrkEdge.h*): Sequence of points, with a start and end node.
- *DtNetworkSegment* (*netwrkSeg.h*): Vectors used in the representation of linear or area features.

DtNetworkSegment is an implementation detail. It is exposed to provide you with the underlying representation of the edge, where necessary, but it is recommended that you use the point interface exposed by the *DtNetworkEdge*.

Network nodes and edges have a *DtSpecification* (*specification.h*) associated with them. The specification holds information about the feature, such as the number of floors it has (in the case of a building), the height and width of the feature, and identifiers, such as DFAD or SEDRIS codes. For more information about specifications, please see "Feature Specifications," on page 3-10.

3.4.1. Linear Features

Linear features, such as roads and rivers are made up of one or more *DtNetworkEdge* objects. A *DtNetworkEdge* is made up of a start *DtNetworkNode*, an end *DtNetwork-Node*, and a series of points that define the smallest straight segments of the edge. (This is why, internally, the edge is represented as both a series of points and as a series of *DtNetworkSegments*.) A *DtNetworkSegment* represents a portion of a network that can be approximated by a line segment (vector).

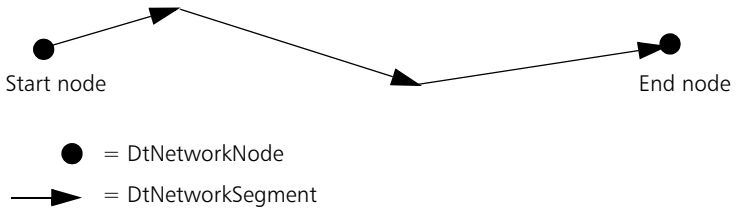

Figure 3-2. Linear feature (edge)

Edges have a direction. The sequence of segments that comprise an edge never intersect one another. A *DtNetworkSegment* never exists by itself in the vector network – it is always part of a *DtNetworkEdge*. Edges can be connected to one another (or themselves) at their endpoints.

It is recommended that you use the point interface, and not access segments directly, as they are considered to be an implementation detail and could change in the future.

Iterating Through the Points in an Edge

To iterate through the points of an edge, ask the edge for a *DtNetworkEdgeTraverser* object. You must specify whether or not to respect the directionality of the edge. If you ignore the directionality, both a forward and backward traverser may be created. However, if you respect the directionality, then for a unidirectional street, only a forward edge traverser can be created.

Once a traverser is created for an edge, ask for a *DtNetworkEdgePointIter* over that edge. The point iterator allows you to iterate over the points contained in the edge. The *DtNetworkEdgePointIter* abstracts away the direction of traversal. Thus, when moving both forwards and backwards along an edge, incrementing the *DtNetworkEdgePointIter* returns an iterator pointing to the next point as appropriate.

3.4.2. Areal Features

Areal features are represented by a *DtNetworkEdge* that forms a closed figure (initial and final point are the same), plus an additional point located approximately in the center of the feature.

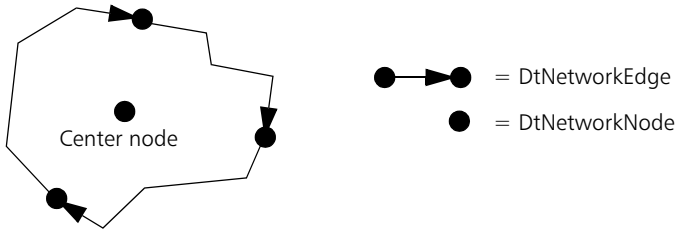

Figure 3-3. Areal feature

Each area has its own specification entry, which must include the extents of a bounding box.

3.4.3. Feature Specifications

A *DtSpecification* represents the set of characteristics associated with a feature in the vector network. A specification can include any number of attributes used to describe the feature. For example, the specification for a building might include the number of floors, while the specification for a river might include width and water level. The *DtSpecificationAttribute* class (*specAttrib.h*) represents the individual attributes held in a *DtSpecification*. A *DtSpecificationAttribute* has a string label and a value. Derived kinds of *DtSpecificationAttribute* are defined for integer, real, and string values.

You can look up an attribute in a specification by its string label. For example, to get the ECC code (a SEDRIS string identifier) for a specification attribute, do the following:

```
const DtSpecificationStringAttribute* group =
  (const DtSpecificationStringAttribute*)spec.attribute("ECC");

if(group)
{
  DtInfo("ECC = %s\n", group.string());
}
```

The ECC code is one of a number of specification attributes typically defined in terrain databases. Please see Chapter 4, *The MÄK Specification Model* for a list of attribute label/value pairs used in MÄK terrain databases.

A *DtSpecification* has some predefined attributes. They are fundamental attributes that can be associated with any feature. The API has methods for setting and retrieving these attributes directly. Table 3-1 lists basic mutator methods.

Table 3-1: Attribute accessor and mutator methods

Accessor/Mutator	Description
setLength() length()	Length of the feature
setWidth() width()	Width of the feature
setColor() color()	RGB color
setVisibility() visibility()	Boolean is visible on/off

The length and width attributes can also be accessed by the GDB_Length and GDB_Width attribute labels.

The *DtSpecification* class allows any number of additional *DtSpecificationAttributes* to be defined and added to it. For example, to create and configure a specification for a building, do the following:

```
DtSpecification spec;

// set values for the predefined attributes

spec.color(DtColor(100,100, 100, 0));
spec.setLength(100);
spec.setWidth(10);

// Add a “Type” attribute to the specification, with a value of “Building”

DtSpecificationStringAttribute sattr;
sattr.setLabel("Type");
sattr.setValue("Building");

spec.addAttribute(sattr);
```

DtSpecification stores the attributes in a hash list indexed by label. To modify a value, add the attribute again with a new value and it will replace the old one.

If you create a *DtSpecification* for an areal vector feature, that is, the specification has a FeatureType = 2 attribute, then the specification must also contain the following attributes:

- DtXMin
- DtXMax
- DtYMin
- DtYMax
- DtZMin
- DtZMax.

Each of these attributes is a double. They represent the extent of the areal feature. If these attributes are not present, VR-inTerra uses 0.0, which may result in unintended consequences.

3.4.4. The MÄK Specification Model

MÄK has developed its own model based on the Digital Feature Analysis Data (DFAD) specification and SEDRIS enumerations. While you can develop your own specification model through the API, the MÄK specification model is already in place (all of our terrain readers use this model). You can use it as-is, extend it, or develop your own scheme for classifying and describing feature data. Please see Chapter 4, *The MÄK Specification Model* for a description of the specification attributes used in the MÄK specification model.

3.4.5. Querying the Vector Network

You can query the vector network to ask for intersections with features. To do this you need to provide a *DtVectorNetworkRecord* (*vecNetwrkRecord.h*) parameter, in which you can specify the criteria for selecting terrain features along the intersection chord. Intersected vector network features that meet the criteria in the record parameter are returned in a *DtVectorNetworkIntersectionRecord* (*vecNwrkIntRec.h*). For example, to query for the terrain skin intersection and for all of the vector network features along the chord, do the following:

```
// Configure vector network record parameter to only select roads
DtVectorNetworkRecord dataRec;
DtSpecificationStringAttribute specAttribute;
specAttribute.setLabel("DtFeatureGroup");// MAK-defined DtGroup attribute
                                  // used to classify features
specAttribute.setValue("MAK010"); // roads
dataRec.addEqualCondAttr(specAttribute);
DtVectorNetworkIntersectionRecord intersectionRec;
// Calculates intersection with terrain geometry and vector network
tdb2.intersect(chord, isectPoint, intersectionTime, &dataRec,
  &intersectionRec);
```

Querying the vector network is a bit different from querying the terrain skin. While the interface presented by both is named intersection, what is actually occurring with the vector network is not intersection, but rather selection based on criteria. As a result, to perform an actual intersection with vector features, the features must be selected from the vector network as described in this section, and then an actual intersection test must be done, using the properties of the features that define an intersectable object, presumably a volume (width, length, height).

3.4.6. Testing for All of the Terrain Intersections Along a Chord

You can perform intersection tests on the terrain database that return all of the points along a given chord that intersect the terrain database. This applies to terrain skin as well as features. Please see *terrainInterface.h* for the set of all available intersection methods.

3.4.7. Using Metrics to Query the Vector Network

In addition to the intersection tests discussed in the previous sections, you can make more customized queries to the vector network using metrics. Metrics provide a mechanism to select elements from the vector network based on:

- Specification requirements: Qualifies a feature if the specification of the feature is valid for the current search. For example, is a road, is not a river, height is greater than zero, and so on.
- Measure: A numeric value used to qualify a feature based on some criteria, such as, distance to a point of interest, length of an edge, and so on.

DtMetric (*metric.h*) is the abstract base class for metrics. It allows feature attributes to be evaluated as equal, different, greater than, and lesser than. You can use the following predefined metrics to query the vector network:

- *DtRangeNetworkNodeMetric* (*rangeNetNodMet.h*) – Network nodes within a given range (for point features such as trees, buildings).
- *DtRangeNetworkSegmentMetric* (*rangeNetSegMet.h*) – *DtNetworkSegments* within a given range (for segments that may be part of linear features, such as a coastline or a road).

DtRangeNetworkNodeMetric and *DtRangeNetworkSegmentMetric* respect the **EvalInZ** parameter. If **EvalInZ** is false, they ignore the Z value in their calculations. If **EvalInZ** is true, they evaluate Z when calculating distance.

- *DtXyNetworkSegmentMetric* (*xyNetSegMet.h*) – *DtNetworkSegments*, sorted by increasing distance from a location, in XY only.

You can use these metrics to evaluate the vector network based on feature specification and range or distance to a point or chord of interest. They calculate intersection against the vector network and calculate paths (in the sense of paths through a graph) using the vector network.

For example, to retrieve all of the trees within range of a location:

```
// Create a range metric - we will use this to select features (nodes)
// from the vector network that fall within range of a location.
DtRangeNetworkNodeMetric nodeMetric;

// Create a specification attribute - we will use this to select
// the types of nodes in our metric. Select only those features having
```

```
// the MAK DtGroup specification of type “MAK040” (vegetation)
DtSpecificationStringAttribute sattr;
sattr.setLabel("DtFeatureGroup");
sattr.setValue("MAK040");

// Set the location of the center of our search for trees, in meters
nodeMetric.setCenterOfInterest(DtPoint(2000, 2000, 0));

// Set the range of the search, in meters
nodeMetric.setMaxRange(45);

// Take EvalInZ attribute into account in distance calculations
nodeMetric.setEvalInZ(true);

// Add a test for the specification attribute (that is, does the feature
// have a DtGroup attribute with a value of "MAK040"?)
nodeMetric.addEqualCondition(sattr);

// Query the vector network for the list of nodes using our metric. It
// returns all of the trees within 45 meters of location (2000, 2000, 0).
DtVectorNetwork* vectorNetwork = terrainInterface()->vectorNetwork();
DtList nodeList;
myVectorNetwork->collectAllNodes(nodeMetric, nodeList);
```

The node list contains pointers to all of the *DtNetworkNode* objects that meet the requirements specified in nodeMetric. The list is sorted in order of closest to furthest from the center of interest. You can add any number of specification attributes to your metric, to further refine your search.

3.4.8. Synchronous and Asynchronous Calls with Paged Terrains

Some terrain types, such as streaming terrain and MetaFlight, utilize paging to limit the amount of terrain loaded by the application at one time. Typically the terrain is broken up into pages along a spatial grid and the pages are loaded on demand when an intersection method requires the data. If the interface was to do nothing special with paged terrains, when an intersection landed in an area where the data was not yet loaded, the system would have to wait for the data to be loaded. In some cases this may take a fairly long time and lead to undesirable results for the simulation. The *DtTerrainInterface* can cope with the paged terrains in two ways to help minimize the occurrence of missed pages and long intersection times.

The first method is if the user can inform the terrain interface of its desire to use an area that is not loaded ahead of time. This is done through the preloadTerrainAreas() method. Calling this method informs the interface of the areas of the terrain that will be required and the interface will start loading those areas in the background. However, there is no guarantee that those areas will be loaded immediately and be available for intersection calls. In this case the user can perform the intersection calls asynchronously.

To make a terrain method asynchronous the caller must provide a Boolean in the intersection method called DataAvailable. When this Boolean is present and the interface is forced to make a call that would cause it to load a page, the method will return immediately and DataAvailable will be set to false. This indicates that the call has neither succeeded nor failed, just that it will take some time to load the page. The caller then can perform other calls and return to this call at a later time when the data may have had a chance to load. When DataAvailable is not present the call will always block, waiting for the call to end no matter how long it takes.

The following code shows a thread that waits for the data to arrive without consuming the cpu during its wait.

```
int maxWait = 10;
double height = 0.;
while (true)
{
  bool dataAvailable = false;
  double height =
  terrainInterface.terrainHeightAboveSeaLevel(localLocation,
    &dataAvailable);
  if (!dataAvailable)
  {
    --maxWait;
    if (maxWait == 0)
    {
      break;
    }
    DtSleep(0.5);
  }
  else
  {
    break;
  }
}
```

The code gives the terrain interface 10 tries and sleeps between each try, allowing other threads to continue. Also note that while the call to terrainHeightAboveSeaLevel() is only different by the inclusion of the DataAvailable boolean, there needs to be a lot more surrounding code to deal with the case when the data is not available.

The terrain interface also has a method called setAssertOnBlockingTerrainCalls(). By setting this value to True, the interface will assert every time a terrain call is made that is called in a non-asynchronous manner. This is useful for testing to find the calls that are made to the terrain that are blocking. Additionally each call can be set to override the blocking assert when the blocking aspect of the call is the required functionality.

3.5. Terrain Implementations

Terrain database files are imported using *DtTerrainImplementations*. A *DtTerrainImplementation* (*terrainImplementation.h*) knows how to import terrain geometry data (terrain skin) from an external format and provide intersection capabilities. VR-inTerra has the following terrain implementations:

- Ellipsoid implementation.
- MÄK Terrain Format (GDB) implementation.
- VR-Vantage implementation.
- Terrain passthrough layers.

These terrain implementors handle the following terrain types:

- Digital Terrain Elevation Data (DTED)
- Shapefile (SHP)
- MultiGen Paradigm OpenFlight (FLT) and MetaFlight
- Compact Terrain Database (CTDB) versions 4b, 7b, and 7l
- VR-TheWorld Streaming Terrain
- Open Scene Graph IVE
- Extent.

For example, to load an OpenFlight database, do the following:

```
// Create and initialize a terrain interface
DtTerrainInterface terrainInterface;
terrainInterface.init();

// Load the terrain database
// Openflight files.
if(!terrainInterface.loadTerrain("HL_94.flt.flt"))
{
  DtWarn("Couldn't load terrain database file\n");
  return 1;
}
```

3.6. Examples

VR-inTerra includes an example, m1a2, which simulates an M1A2 tank driving on terrain and making terrain queries. Please see the comments in the source files for details.

The MÄK Specification Model

This chapter describes the specification model used for feature data.

4.1. The MÄK Specification Model

When feature data is imported into MÄK GDB format, a particular set of attribute label/value pairs are always assigned to feature specifications. These attributes include DFAD codes and SEDRIS enumerations. The rationale behind this approach is that there is a considerable amount of data already available in the DFAD format, and the SEDRIS enumerations are flexible and extensible.

The MÄK specification model also includes a classification called *DtFeatureGroup*. The DtFeatureGroup attribute organizes features into one of several broad categories. A corresponding DtFeatureGroupName attribute is also assigned to each feature. It contains a human-readable string representation of the *DtFeatureGroup* value. Features are categorized according the attributes listed in Table 4-1.

Table 4-1: Feature attributes

DtFeatureGroup	DtFeatureGroupName
MAK000	Feature
MAK010	Road
MAK020	Railroad
MAK030	Waterway
MAK040	Vegetation
MAK050	Power and Communications
MAK999	Other Feature

When you import vector data into a *DtTerrainInterface* (from DFAD, DFD, SHP, or C7B sources) a vector network is created that uses the MÄK specification model. The VR-inTerra takes advantage of this design. During the import process, it reads the following files and uses them to specify how incoming DFAD and SEDRIS codes should be mapped to various feature attributes:

- *dfad_sedris_map.txt* – maps each type of DFAD/DFD feature to a default SEDRIS feature.
- *importFile.txt* – defines the list of DFAD/DFD features to import into the GDB file. Each possible feature specifies IMPORT or NOIMPORT.
- *sedris_colormap.txt* – defines the color representation for each subcategory in the SEDRIS code, in the format RGBA.
- *FID_SMC.txt* – defines the list of FID and SMC codes for DFAD and DFD basic features.

4.2. MÄK Specification Attributes

Table 4-2 lists attributes that MÄK uses in its specification model that are always present for each feature in the vector network.

Table 4-2: Specification model attributes

Label	Type	Description
GDB_Width	real	Width of the feature. ◆ Point – If the width > 0, actual width ◆ Point – If the width = 0, a circular feature with radius length ◆ Linear – actual width ◆ Areal – 0 (not used).
GDB_Length	real	Length of the feature. ◆ Point – actual length ◆ Linear and Areal – 0 (not used).
DtOrientation	integer	Rotation about the database Z-axis. Starts with a value of 0 at the X-axis and increases in the counter-clockwise direction. ◆ Point: 0 – 360 degrees.
DtHeight	integer	Predominant elevation in meters.
DtFeatureType	integer	Defines the "shape" of the feature. ◆ 0 – Point ◆ 1 – Linear ◆ 2 – Areal.
DtFidCode	integer	DFAD code for that type of feature. Please see *./data/importDfad/FID-SMC.txt*.
DtFeatureName	string	DFAD definition for corresponding DtFidCode code. Please see *./data/importDfad/FID-SMC.txt.*
DtSmc	integer	Soil type based on SMC codes. Please see *./data/importDfad/FID-SMC.txt.*
DtFeatureGroup	string	Defines a grouping based on a MÄK extension of SEDRIS (see Table 4-1): ◆ MAK000 ◆ MAK010 ◆ MAK020 ◆ MAK030 ◆ MAK040 ◆ MAK050.

Table 4-2: Specification model attributes

Label	Type	Description
DtFeatureGroupName	string	String name of DtGroup (see Table 4-1): ◆ Feature ◆ Road ◆ Railroad ◆ Waterway ◆ Vegetation ◆ Power and Communications.
ECC	string	Main SEDRIS identification code mapped from DFAD. Please see *./data/importDfad/dfad_sedris_map.tx.*
DtDescription	string	Main SEDRIS identification code description. Please see *./data/importDfad/dfad_sedris_map.txt.*
DtSedrisCodes	integer	Number of additional SEDRIS codes specifying the feature. Please see *./data/importDfad/dfad_sedris_map.txt.*
SEDRISCODE#	string	Where 1 < = # < = DtSedrisCodes. Please see *./data/importDfad/dfad_sedris_map.txt.*
i-SEDRIS code	string	i-SEDRIS code description. Please see *./data/importDfad/dfad_sedris_map.txt*.

Table 4-3 lists commonly used attributes that are not necessarily present in every specification.

Table 4-3: Commonly used attributes

Label	Type	Description
DtFacNumber	integer	Number of the feature, used to provide unique specification for a feature
The following optional attributes apply to area features only:		
DtStructsPerKm	Integer	Number of structures per kilometer squared.
DtPctRoofCoverage	Integer	Percentage of areal covered.
DtPctTreeCoverage	Integer	Percentage of area covered with trees.
DtXMin	Real	Minimum X coordinate of bounding outline, in meters.
DtYMin	Real	Minimum Y coordinate of bounding outline, in meters.

Table 4-3: Commonly used attributes

Label	Type	Description
DtZMin	Real	Minimum Z coordinate of bounding outline, in meters.
DtXMax	Real	Maximum X coordinate of bounding outline, in meters.
DtYMax	Real	Maximum Y coordinate of bounding outline, in meters.
DtZMax	Real	Maximum Z coordinate of bounding outline, in meters.

MÄK implements this specification model using the following rules:

- Each area has its own specification.
- All of the segments, nodes, and edges making up a feature such as a road or river must share the same specification.
- If two features are different by even a single attribute value, they have different specifications.

Using the DtFacNumber attribute, each feature can have its own unique specification. Another way to achieve unique specifications is to add additional attributes, such as street name or river name.

MÄK Products Glossary

This glossary defines terms used in MÄK product documentation.

absolute dead reckoning

A method of dead-reckoning in which an application uses the time stamps contained within state updates if the sender is using absolute time stamping. Contrast with relative dead reckoning.

aggregate

The combination of individual entities, aggregates, or both into a single object. For example, an organizational group such as a platoon.

AMO

See Application Management Object.

Application Management Object

An Application Management Object (AMO) is a standard TENA object that is supposed to be used by every TENA application. It gives other applications information about the local application. The VR-Link exercise connection, by default, publishes an AMO object, updates it at a specified rate (the rate is part of the AMO's state repository; the default is 30 seconds), and updates the number of objects published and proxies (TENA reflected objects) held.

API

Application Programming Interface.

articulated part

A part of an entity that is capable of movement relative to the entity, such as a turret or gun.

attached part

An object that is attached to another object, such as a rocket attached to a jet.

body coordinate frame

A coordinate framework centered on an entity. To represent the orientation of an entity, an application uses Euler angles or a rotation matrix that rotates from a reference frame to the body coordinate frame.

channel

A channel renders the view of an observer in a 3D visualization application.

clipping

A feature that prevents the display of terrain and objects or parts of objects, that are closer than or farther than specific distances from the observer.

clipping planes

The range of distances from the observer (near clipping and far clipping) in which an application clips objects.

culling

The process of discarding database objects which are not within view, and therefore need not be rendered and displayed.

Cartesian coordinates

A system for indicating location by means of three planes intersecting at right angles to one another at the origin.

Data Distribution Management

The set of HLA services used to specify the routing of data between publishers and subscribers. DDM adds greater control over the Declaration Management (DM) services that only specify filtering based on the class of the data (e.g., ground vehicle versus aircraft). DDM uses the data content to express regions of interest and regions of affect whose overlap establish a communication channel. An example is geographic filtering where a publisher indicates a region around the entity or effect and a subscriber indicates a region expressing its sensor range.

dead-reckoning

A process by which an application calculates the expected location of an entity during periods between state updates, based on velocity, acceleration, and rotational velocity.

DDM

Data Distribution Management.

Defense Modeling and Simulation Office

The former name of the executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provided a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. Renamed Modeling and Simulation Coordination Office. Currently the MSCO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness.

DIS

Distributed Interactive Simulation.

DIS data packets

See Protocol Data Unit.

display engine

An object in an application that can render 3D data.

Distributed Interactive Simulation

The DIS protocol is a set of standards that govern how participating applications share information about the virtual world. The protocol specifies a set of packets, called protocol data units (PDUs), that communicate this information. Each PDU identifies the sender and contains other information depending on the PDU type. The DIS protocol also specifies when and how frequently PDUs are sent.

The DIS protocol has been superseded by the High-Level Architecture for use by the Department of Defense.

DMSO

See Defense Modeling and Simulation Office.

emitter

A simulated device on an entity that emits electromagnetic radiation, for example, radar.

entity

An element in a simulation, such as a vehicle or a person, that is represented in the simulation through the issuance of state messages.

entity maintenance

A process by which the MÄK Data Logger compensates for discontinuities following a time jump. The Logger sends out interim state messages for entities that were present before the time jump so that they do not time out before the next update message arrives.

entity-type

A list of seven components as specified by the DIS protocol and the HLA RPR FOM. If none of the seven components contains a wildcard (-1) value, then the entity type refers to a specific, narrowly-defined entity. If some components contain a wildcard, then the entity type refers to a class of entities. A larger number of wildcards indicates a broader, more general class.

The components of an entity-type are:

- Entity kind
- Domain
- Country
- Category
- Subcategory
- Specific
- Extra.

environmental process

According to the IEEE 1278.1a specification (DIS), environmental process PDUs communicate simple environment variables, small scale environmental updates, and embedded processes.

Euler angles

A set of three angles used to describe the orientation of an entity as a set of three successive rotations about three different orthogonal axes (x, y, and z). These angles specify successive rotations needed to transform from a reference coordinate system to the entity's body coordinate system.

event

An interaction between objects or between an object and the terrain, such as firing of a munition, or a collision of entities.

execution

The TENA term for one or more interacting simulation applications.

Execution Manager

An Execution Manager governs a TENA execution for applications joining, resigning, or changing subscriptions to that execution.

exercise

The DIS term for a one or more interacting simulation applications. Compare to federation execution in HLA.

exercise connection

The object (*DtExerciseConn*) through which a VR-Link application connects to the network. State messages are sent through the exercise connection and information about remote entities is received through the exercise connection. (All MÄK products are VR-Link applications.)

exercise ID

A numeric identification for a DIS simulation exercise.

FED file

federate

A connection to the RTI. Typically a single simulation application can be thought of as a federate.

federation

A group of HLA federates capable of playing in the same federation execution.

Federation Object Model

Defines the data content of a federation execution.

federation execution

The federation execution represents the actual operation, over time, of a subset of the federates and the RTI initialization data taken from a particular federation. A federation execution is the logical equivalent of a DIS simulation exercise.

field of view

Controls the perspective of the observer.

A wide field of view creates an effect like that of a wide-angle camera lens. Objects appear smaller and farther away from the observer, since the observer coverage spans a wider area. Depth become exaggerated.

A narrow field of view creates an effect like that of a telephoto lens. Objects appear larger and closer to the observer, and the overall scene depth appears flattened. The distances between objects appears compressed.

filter range

A setting that prevents distant entities from being processed while allowing distant terrain to appear normally.

FOM

See Federation Object Model.

FOM Module

In HLA Evolved, defines additional modular data content for a federation execution, usually extensions to an existing FOM. (Also supported by the HLA 1.3 and HLA 1516 versions of the MÄK RTI. Per the HLA Evolved interface specification:

"A partial FOM (containing some or all OMT tables) that specifies a modular component of a FOM. A FOM module may contain classes not inherent to it but upon which the FOM module depends, i.e., superclasses to the modular components. These superclasses will be included in the FOM module either as complete or scaffolding definitions."

frame rate

The rate at which the application displays updated images.

ground clamping

A process by which the a 3D visualization application keeps an entity anchored to the surface of its terrain database, regardless of the altitude data contained in its entity state message.

geocentric coordinates

A coordinate system calculated with respect to the earth's center. The origin of the geocentric coordinate system is the center of the earth. The positive X-axis passes through the prime meridian at the equator; the positive Y-axis passes through 90 degrees east longitude at the equator; and the positive Z-axis passes though the north pole.

geodetic coordinates

A coordinate system in which position is determined relative to a reference ellipsoid, such as the surface of the earth at sea level. In MÄK applications, geodetic coordinates consist of latitude and longitude in radians, and altitude in meters above the reference ellipsoid.

gridded data

Data that has been processed in a rectangular array of points, in X, Y or latitude/longitude, at which single data values define a two dimensional function. According to the IEEE 1278.1a specification, gridded data transmits information about large-scale or high-fidelity spatially and temporally varying ambient fields and about environmental processes and features.

guise

An alternative entity type used to display an object depending on the force ID. For example, a tank could look like an M1A1 to friendly forces and a T72 to hostile forces.

head-up display

A set of indicators and readouts superimposed onto a graphics display. Also called an overlay.

heartbeat

In DIS, the frequency with which current PDUs are sent to the network regardless of whether or not the entity's state has changed.

High-Level Architecture

The High Level Architecture (HLA) for simulations is a U. S. Department of Defense (DOD)-wide initiative to provide an architecture to support interoperability and reuse of simulations. The HLA supersedes DIS for the DOD.

HLA

See High-Level Architecture.

interaction

A message describing a simulation event. An interaction describes an event, it does not update an object's state.

Logger control PDUs

logical range

A suite of TENA Resources, sharing a common object model, that work together for a given range event. Similar in concept to the HLA Federation.

Local RTI Component

The libraries on a computer that implement an RTI. A federate communicates with the HLA network through the LRC.

Logical Range Object Model

The data definition file for TENA exercises. It contains the set of object and message definitions that may be used in the exercise.

LRC

See Local RTI Component.

LROM

See Logical Range Object Model.

MÄK Technologies Lisp

An adaptation of the Lisp language used in configuration files for MÄK products.

message

A general term used to refer to DIS PDUs and HLA interactions and state updates. In TENA, a message is similar to an HLA interaction.

MIM

The MOM & Initialization Module (MIM) allows for extensions to be made to the HLA standard MOM. It is a subset of the FOM that contains OMT tables that describe the HLA MOM. The MIM also contains additional predefined HLA constructs such as object and interaction roots, data types, transportation types and dimensions. HLA specifies a standard MIM that is incorporated into all FDDs automatically by the RTI. The standard MIM can be replaced with a user-supplied MIM containing the standard MIM plus extensions.

Modeling and Simulation Coordination Office

The executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provided a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. The MSCO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness. (Formerly DMSO)

MSCO

Modeling and Simulation Coordination Office. (Formerly DMSO.)

MTL

See MÄK Technologies Lisp.

Network Naming Service

Acts as a registry for TENA Execution Managers.

NNS

See Network Naming Service.

orientation clamping

Adjusts the pitch and roll of an entity so that it appears properly seated when the terrain is inclined. For example, if a tank is moving horizontally across the face of a hill, orientation clamping prevents the tank from appearing level, and therefore, partially embedded into the hillside. Used with ground clamping.

object

An element in a simulation that has persistence, as opposed to an interaction, which is a transient element.

object handle

An integer that an application uses to identify a particular object in RTI service calls. An object handle is meaningful only to a particular federate. The same object can be known to different federates by different object handles.

object name

A character string that can be used to identify an object. In HLA, the object name is known to the RTI, and the RTI provides functions to find out an object's name, given its handle, and vice versa. Object names can be chosen by applications that register the objects with the RTI, however if you do not want to choose names for objects, the RTI will assign names for you.

observer

In MÄK 3D applications, the point of view into the scene. (Called the eyepoint in MÄK Stealth 6.x and earlier.)

PDU

See Protocol Data Unit.

Protocol Data Unit

A unit of data message (packet) that is passed on a network between DIS simulation applications.

proxy

In TENA, a reflected object (Stateful Distributed Object).

radio transmitter

A simulated device on an entity, capable of transmitting radio communications.

radio receiver

A simulated device on an entity, capable of receiving radio communications.

range resource

A participant in a TENA execution (similar in concept to the HLA federate).

Realtime Platform Reference FOM

An HLA reference FOM based on the DIS protocol.

recording

A Data Logger file that stores a history of the interactions in a simulation for playback.

reflected entity list

A list of entities simulated by remote applications (federates in HLA) and about which the local simulation has received information over the network.

reference FOM

A FOM designed to be used as a whole, or with modification, by a wide variety of similar federation executions.

relative dead reckoning

A method of dead reckoning in which an application approximates the location of an object based on the local time of receipt of the last state update message.

remote display engine

A display engine running in an application that is separate from the master application, and which is controlled by the master application.

RID file

See RTI Initialization Data.

RTI Initialization Data

The initialization data required by the RTI for operation.

Data required by an RTI during initialization, independent of the FOM being used. RID data is usually dependent on a specific implementation of the RTI.

RPR FOM

See Realtime Platform Reference FOM.

RTI

See Run-Time Infrastructure.

Run-Time Infrastructure

A library and other supporting software that implements the HLA interface specification. All federates communicate with one another in an HLA environment through RTI functions.

SDO

See Stateful Distributed Object.

servant

A published object (SDO) in TENA.

Simulation Interoperability Standards Organization

A group that seeks to promote modeling and simulation interoperability and reuse for the benefit of diverse M&S communities, including developers, procurers, and users, world-wide.

simulation time

In VR-Forces, simulation time is used in dead-reckoning of remote entities and thresholding of local entities. Typically, simulation time is set once during each iteration of the application's main simulation loop so that all entities are dead-reckoned based on the same value of current time.

SISO

See Simulation Interoperability Standards Organization.

smooth period

The period of time over which trajectory smoothing takes place.

smoothing

A method of ensuring that transitions from an entity's dead-reckoned position to its actual position are not so abrupt as to be visually disconcerting.

state

The current status of an object, including location, direction of movement, extent of damage, and so on.

Stateful Distributed Object

An object in a TENA exercise.

tape

An alternative term for a Logger recording. Not a physical magnetic tape.

TENA

See Test and Training Enabling Architecture.

TENA Middleware

terrain following

In a 3D visualization application, causes the observer (eyepoint) to maintain a constant distance above the terrain surface. The observer's height changes in tandem with the peaks and valleys as it passes over the geography.

Test and Training Enabling Architecture

Test and Training Enabling Architecture (TENA) is an interoperability architecture used to pass data in the form of object updates and messages (like HLA objects and interactions) from one application to another.

timeout

The period of time in which an application continues to display an entity after that entity's update messages have stopped appearing on the network. Time-outs are usually not used in HLA because there is no set frequency (no heartbeat) for transmitting messages.

timescale

A factor by which time is accelerated or slowed during playback of a Data Logger recording.

topographic coordinates

A right-handed Cartesian coordinate system whose X-Y plane is tangent to the earth's surface at the origin, with the positive X axis pointing north, the positive Y axis pointing east, and the positive Z axis pointing down. There are an infinite number of topographic coordinate systems – one for each point on the earth's surface.

topographic coordinate frame

A coordinate frame in the context of the terrain.

trajectory smoothing

A method used by to smooth positional discontinuities that could occur when new state updates arrive.

UDP port

A network channel through which an application sends and receives data for DIS exercises.

Universal Transverse Mercator

In general, a non-cartesian coordinate system in which the X, Y, and Z correspond to easting (nearly east), northing (nearly north), and height above an ellipsoid that approximates the surface of the earth.

UTM

See Universal Transverse Mercator.

view control messages

A set of programmatic messages that let you control the VR-Vantage remotely.

view floor

In MÄK Stealth 6.x and earlier, a minimum height above the terrain in Absolute or Track mode, below which the observer is not allowed to go. The view floor is measured relative to the terrain surface.

Index

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

X-Y-Z

Link - Simulate - Visualize

VRI-1.0-1-110805

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