VR-Vantage

Developers Guide

VT MÄK
A company of VT Systems

VR-Vantage

Developers Guide

VT MÄK
A company of VT Systems

VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

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

info@mak.com

www.mak.com

Revision VRV-1.4-2-111218

Contents

Preface

Chapter 1 The VR-Vantage Toolkit

Chapter 5 Drivers

Chapter 6 Composing Scenes

Chapter 10 Building VR-Vantage Applications

Chapter 11 The VR-Vantage Control Toolkit

Preface

This manual is for persons who will use the VR-Vantage API to modify the VR-Vantage applications or build new applications with the VR-Vantage Toolkit. The manual assumes that you are familiar with C++, IDEs, and basic application development tasks.

For the latest product information, please see release-specific documentation for your version of the VR-Vantage.

How the Manual Is Organized

This manual is organized as follows:

Chapter 1, *The VR-Vantage Toolkit*, is a general introduction to the VR-Vantage toolkit. It also describes the licensing restrictions of third-party libraries.

Chapter 2, *VR-Vantage Application Architecture*, presents a high-level view of the VR-Vantage architecture and shows how to create a simple VR-Vantage application.

Chapter 3, *GUI Architecture* describes the architecture of GUI elements and covers other implementation details.

Chapter 4, *The Display Engine*, provides details about the display engine, which is the core component of any VR-Vantage application.

Chapter 5, *Drivers*, describes how VR-Vantage uses drivers to control objects.

Chapter 6, *Composing Scenes*, describes the objects that make up a VR-Vantage scene.

Chapter 7, *3D Graphics*, describes how 3D graphics are rendered.

Chapter 8, *2D Graphics* explains how VR-Vantage renders 2D graphics.

Chapter 9, *VR-Vantage Design Patterns*, describes the design patterns used for much of the code in VR-Vantage.

Chapter 10, *Building VR-Vantage Applications*, explains how to link and compile VR-Vantage applications. It describes the libraries that VR-Vantage uses and other details of the build process.

Chapter 11, *The VR-Vantage Control Toolkit*, explains how to use the VR-Vantage Control Toolkit to send view control PDUs to VR-Vantage applications. It also explains how to use the VR-Vantage Remote Draw API, which lets remote applications draw various kinds of geometric shapes and text on the VR-Vantage scene

Appendix A, *Menu Names* lists the menu name codes that you need to know to modify VR-Vantage menus.

Documentation Set

VR-Vantage documentation is provided in Adobe Acrobat PDF format. Documents are in *vrvantagex.x/doc*. On Windows, the documentation is accessible from the VR-Vantage folder on the Start menu. The VR-Vantage documentation set is as follows:

- *VR-Vantage Users Guide* describes how to use the VR-Vantage applications.
- *VR-Vantage Developers Guide* explains how to use the VR-Vantage API.
- *VR-Vantage Release Notes* lists system requirements, release-specific requirements, new features and updates, bug fixes, and known problems.
- Class reference documentation in HTML format.

MÄK Products

VR-Vantage 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-Vantage 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/get-licenses.html
Product version and platform information:	www.mak.com/support/product-versions.html
For the free, unlicensed MÄK RTI:	www.mak.com/resources/bonus-material/cat_view/16-bonus-materials/24-mak-high-performance-rti.html
MÄK Community Forum:	www.mak.com/community-forum/1-forum.html

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.
Blue text	A hypertext link to another location in this manual or another manual in the documentation set.
Blue bold text	A hypertext link to class documentation.
`{ }`	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-Vantage home directory. For example, the directory *vrvantagex.x/doc* appears in the text as *./doc*.

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://source-ware.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 plat-forms. 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

Third-Party Licenses for VR-Vantage Applications

VR-Vantage applications use a variety of third-party libraries. Developers who want to use these libraries may be required to purchase developer's licenses. Please see "Third-Party Software and Content," on page 1-3 for details.

1

The VR-Vantage Toolkit

This chapter introduces the VR-Vantage toolkit and describes the third-party libraries it uses and their licensing requirements.

1.1. Introduction

The VR-Vantage product line consists of the VR-Vantage Toolkit and three application built with the Toolkit: VR-Vantage Stealth, VR-Vantage IG, and VR-Vantage XR. The VR-Vantage applications are end-user graphical user interfaces (GUIs) for viewing simulation in 3D and 2D. They are described in *VR-Vantage Users Guide*.

The VR-Vantage Toolkit comes with all the software you need to completely rebuild visual applications like VR-Vantage Stealth or VR-Vantage IG. It also allows you to customize and extend these applications to fit your unique requirements. The VR-Vantage Toolkit also gives you the power to embed any of the VR-Vantage GUI capabilities directly into your simulation applications. The VR-Vantage Toolkit is based on OpenSceneGraph[1], so you can leverage value-added plug-ins built by the OSG community and MÄK partners.

VR-Vantage supports rendering to multiple channels running on separate computers through an application called the VR-Vantage Display Engine. You can run display engines on every machine on which you want to render a channel, and connect to them using the VR-Vantage applications, or a custom-built application, to render the same scene on several machines at once.

1.1.1. VR-Vantage Control Toolkit and Remote Draw API

The VR-Vantage Control Toolkit (VCT) lets you send DIS or HLA messages that can remotely control a VR-Vantage application. It supports the VR-Vantage attach modes and most visual states, such as track histories, trailing effects, entity labels, and so on. For details, please see "The VR-Vantage Control Toolkit," on page 11-3.

The VR-Vantage Remote Draw API allows third party applications to create, send, and manage 2D and 3D objects to be drawn by VR-Vantage. For details, please see "The VR-Vantage Remote Draw API," on page 11-5

1. "OpenSceneGraph is an Open Source, cross-platform graphics toolkit for the development of high-performance graphics applications such as flight simulators, games, virtual reality and scientific visualization. It is based on the concept of a SceneGraph, providing an object-oriented framework on top of OpenGL." (http://www.openscenegraph.org/projects/osg/wiki/About/Introduction)

1.2. Third-Party Software and Content

The VR-Vantage product includes software and content licensed from third parties, including:

- SilverLining™: real-time sky and 3D cloud rendering from Sundog Software
- GL Studio®: interactive cockpit instrumentation and HMI from DiSTI®
- DI-Guy™: realistic human character animation from Boston Dynamics
- SpeedTree®: animated, 3D foliage from Interactive Data Visualization (IDV)
- 3D models, terrain, and graphical content from Simthetiq, RealDB, TurboSquid, Terrex (now Presagis), and TerraSim
- OpenSceneGraph: an open source 3D graphics toolkit hosted at http://www.openscenegraph.org
- osgEarth: an open source streaming terrain plug-in by Pelican Mapping, at http://osgEarth.org.

The run-time and developers' rights for VR-Vantage customers vary from vendor to vendor. Please read this section carefully to understand which rights are included with VR-Vantage licenses, and which rights require additional third-party licenses.

1.2.1. SilverLining

VR-Vantage uses SilverLining software and content to compute lighting and to render the sky, clouds, sun, and moon. SilverLining is developed by Sundog Software (http://www.sundog-soft.com).

Most VR-Vantage customers do not need to buy a SilverLining license of any kind. In general, a VR-Vantage customer has the right to use the SilverLining technology that is built into VR-Vantage in any VR-Vantage-based application (including custom applications built using the Toolkit). However, there are two exceptions:

- If you want to extend or change the way that MÄK has integrated SilverLining into VR-Vantage by writing directly to the SilverLining API, then you need a SilverLining Developer's License.
- If you are building a commercial video game or other "mass-market" (consumer-level) commercial product on top of the VR-Vantage Toolkit, and the resulting product includes the SilverLining functionality, then you need a SilverLining Developer's License.

SilverLining libraries, textures and other graphics resources are distributed in the VR-Vantage package solely so that VR-Vantage can use them. Use of SilverLining software or content outside of VR-Vantage-based applications is not permitted by the VR-Vantage license.

1.2.2. GL Studio

VR-Vantage uses GL Studio software and content to render interactive cockpit instrumentation displays. GL Studio is developed by DiSTI (http://www.disti.com).

VR-Vantage is delivered with several generic cockpit displays. You can use the built-in cockpit displays in any VR-Vantage-based application (including custom applications built using the Toolkit), without a GL Studio license of any kind.

If you want to build custom cockpits using the GL Studio editor or GL Studio API, you need a GL Studio Developer's license. If you want to execute custom cockpits in VR-Vantage-based applications (regardless of whether the custom cockpits are built by you or a third party), you need GL Studio Run-Time licenses.

GL Studio libraries and graphics resources are distributed in the VR-Vantage package solely so that VR-Vantage can use them. Use of GL Studio software or content outside of VR-Vantage-based applications is not permitted by the VR-Vantage license.

1.2.3. DI-Guy

VR-Vantage uses DI-Guy software and content for human character animation. DI-Guy is developed by Boston Dynamics (http://www.bostondynamics.com).

VR-Vantage comes with DI-Guy functionality built-in and with a large set of DI-Guy characters, appearances, and animations. A VR-Vantage customer does not need a DI-Guy license of any kind to use DI-Guy animated characters in a VR-Vantage-based application (including custom applications built using the VR-Vantage Toolkit).

However, you need a DI-Guy Developer's license if you want to:

- Develop new character types, or alter the geometry, appearances, textures, or motions of DI-Guy characters.
- Directly access the DI-Guy API to modify or extend the way that VR-Vantage uses the DI-Guy API or access DI-Guy functionality that MÄK has not already integrated into VR-Vantage.

DI-Guy libraries, models, textures, motion files, and other graphics resources are distributed in the VR-Vantage package solely so that VR-Vantage can use them. Use of DI-Guy software or content outside of VR-Vantage-based applications is not permitted by the VR-Vantage license.

1.2.4. SpeedTree

VR-Vantage uses SpeedTree software and content for animated real-time 3D foliage and vegetation. SpeedTree is developed by Interactive Data Visualization (IDV) (http://www.speedtree.com).

A VR-Vantage customer does not need a SpeedTree license of any kind to use the SpeedTree functionality that is built into the VR-Vantage applications that MÄK delivers. You may build and run plug-ins to our out-of-the-box applications without a SpeedTree license, as long as your plug-ins do not need to directly access the SpeedTree API.

However, you need a SpeedTree Developer's license if you want to:

- Write a plug-in that uses the SpeedTree API directly to extend or modify the way that SpeedTree is integrated into VR-Vantage.
- Use the VR-Vantage Toolkit to develop new applications, and you want your applications to include SpeedTree functionality.

SpeedTree libraries, models, textures, and other graphics resources are distributed in the VR-Vantage package solely so that VR-Vantage can use them. Use of SpeedTree software or content outside of VR-Vantage-based applications is not permitted by the VR-Vantage license.

1.2.5. 3D Models, Terrain, and Graphical Content

VR-Vantage includes a rich set of 3D models for vehicles, weapons, cultural features and urban clutter (signs, barriers, lampposts, and so on). It also includes several sample terrain databases to help you get started with VR-Vantage, and to help demonstrate VR-Vantage. Much of this content is derived from 3D data that we have licensed from third parties:

- Many of the high-quality vehicle models come from Simthetiq (http://www.simthetiq.com).
- Some models come from Real DB (http://www.realdb.qc.ca/company)
- Some of the middle-eastern building models and urban clutter objects that are used in our sample terrain databases were licensed from Turbosquid (http://www.turbosquid.com)
- The TerraSim_SampleUrban terrain database comes from TerraSim (http://www.terrasim.com)
- The HarrisAfghanVillage terrain database was created using source data from Harris Inc.
- The Berkeley Database was developed by Terrex (now Presagis) (http://www.presagis.com).

All of this content is distributed in the VR-Vantage package solely so that VR-Vantage can use it. Use of any of the VR-Vantage models, textures, terrains, or other content outside of VR-Vantage-based applications is not permitted by the VR-Vantage license.

1.2.6. OpenSceneGraph

VR-Vantage uses OpenSceneGraph for its underlying scene graph representation, rendering, file loading, and many other capabilities. OpenSceneGraph is an open source, cross-platform graphics toolkit for the development of high-performance graphics applications. The OpenSceneGraph source repository is maintained by Robert Osfield at http://www.openscenegraph.org/projects/osg. It is distributed under the OpenSceneGraph Public License (OSGPL), which is based on the GNU Lesser General Public License (LGPL). MÄK will provide modified source code for OSG upon request, as per the OSGPL license.

1.2.7. osgEarth

VR-Vantage uses osgEarth to import streaming terrain elevation and imagery data. The data can be streamed from external servers and sources or from a directory on the computer running VR-Vantage. osgEarth is an open source plug-in to OpenSceneGraph, maintained by Pelican Mapping at http://osgEarth.org. osgEarth is distributed under the GNU Lesser General Public License (LGPL). MÄK will provide modified source code for osgEarth upon request, as per the OSGPL license.

1.3. VR-Vantage and VR-Link

VR-Vantage relies on VR-Link to handle DIS and HLA networking. A VR-Vantage developer's license includes a license for VR-Link so that you can use VR-Link functions and classes directly from applications that use any of the VR-Vantage APIs.

VR-Link provides a protocol-independent interface to DIS and HLA. If you use it, you do not need to know the lower level details of these protocols to build VR-Vantage applications. VR-Vantage applications can take advantage of VR-Link's tools for managing objects and interactions, such as reflected entity lists.

If you are not already familiar with VR-Link, we recommend that you review the conceptual material in *VR-Link Developers Guide* and the description of the protocol-independent interface.

1.4. The Qt Toolkit

VR-Vantage uses the Qt™ Toolkit from Qt Software to build the graphical user interface (GUI). For more information, please see "The Qt Toolkit," on page 10-4.

1.5. Example Projects

VR-Vantage has many example projects to help you understand various aspects of using the VR-Vantage Toolkit. For a complete list, please see the Examples page in the class documentation.

2

VR-Vantage Application Architecture

This chapter describes the VR-Vantage toolkit architecture, introduces key classes, and shows how to create a simple VR-Vantage application.

2.1. Introduction

The VR-Vantage Toolkit includes the classes and functions needed to build or extend 3D visualization applications. It gives you access to almost all of VR-Vantage's functionality, including:

- Windows and channels
- Scene objects such as entities, props, terrain, and the environment
- Observers
- Drivers
- The graphical user interface (GUI)
- The top-level objects and application framework that tie all of the functionality together.

The VR-Vantage applications are built using the VR-Vantage Toolkit. You can use the VR-Vantage Toolkit to customize or extend the functionality of these applications – either by creating plug-ins, or by starting with our sample code and building your own similar applications. You can also use the Toolkit to embed pieces of VR-Vantage functionality into your own applications. This chapter provides an introduction to the VR-Vantage Toolkit. It introduces the ***DtDe*** class – the top-level class in the VR-Vantage Toolkit, and describes the various ways in which your code can interact with it. ("De" stands for display engine.)

Header files for VR-Vantage Toolkit classes have the same name as the class that they define. For example, the header file for *DtDe* is *DtDe.h*. Therefore, in this manual, we do not specify a header file when we introduce a class.

Subsequent chapters describe the other key elements of the VR-Vantage Toolkit – the classes that represent scene objects, drivers, and so on. The *DtDe* class uses these classes to perform most of its work. You will typically access these classes through the *DtDe*. And when you want to extend or customize these classes (to extend or customize the behavior of VR-Vantage applications), you will typically register your subclasses with VR-Vantage through the *DtDe*.

2.2. Extending the VR-Vantage Applications

If you want to modify or extend a VR-Vantage application, there are two main approaches you can take:

- Build a plug-in that can be loaded by these applications
- Create a new application based on one of the supplied examples (*./appsrc/user-Stealth*, *./appsrc/userIG*, *./appsrc/userXR*, or *./appsrc/userPVD*).

In most cases, if your goal is just to extend our applications, we recommend the plug-in approach. Most of our examples demonstrate how to extend our applications through plug-ins. If your goal is to create a new visualization application that shares the same basic framework as a VR-Vantage application, then it will probably make more sense to start with the example projects in *./appsrc* and make changes within the application.

A good way to decide which approach to take is to ask yourself if you would describe the goal of your work as "VR-Vantage Stealth, with the XYZ feature disabled, and the ABC feature added". If the answer is yes, you should probably use a plug-in. If no, then create a new application.

For typical application extensions, plug-ins have the following advantages:

- A plug-in developed with the VR-Vantage Toolkit will work in MÄK's VR-Vantage applications and most other VR-Vantage-based applications. So if your organization is using the VR-Vantage applications in different configurations or doing customized application development, a plug-in can be reused in multiple settings with no additional work.
- Plug-ins facilitate easy integration with the work of other developers. If multiple developers add different capabilities through plug-ins, it is likely that you can use some, or all of them together, in MÄK's VR-Vantage applications without additional development. On the other hand, if each of the developers added features by building separate custom applications, you would have to do additional work to merge the various extensions into one application.
- Plug-ins allow you to easily include or exclude a feature to meet the needs of a particular project. If you do not want to use a feature, do not load the plug-in that implements it.
- It will usually be easier to port a plug-in to a new version of VR-Vantage than it would be to port a custom application. It is less likely that you will have to merge project settings, or merge any source code, if the user*application* example source code changes.
- Using plug-ins may help you avoid the need to purchase licenses for third-party libraries. Some of the third-party libraries that we have licensed (notably SpeedTree), require that developers buy a third-party developers license if they are building new applications using the VR-Vantage Toolkit, but do not require a developers license if a customer is modifying our existing applications using plug-ins.

Ideally, a plug-in module should be limited to a specific, single piece of functionality. For example, if you want to add a new menu to the GUI, and an unrelated new kind of scene object, use separate plug-ins. However, if the new features work together, such as a new kind of scene object and a menu option to configure or control it, then these two pieces of code should probably be built into the same plug-in.

Regardless of whether you choose the plug-in approach or the new application approach, the code you write to implement your new functionality will be largely the same. The principal difference in code between the two approaches is how you access the *DtDe* to register your new classes:

- In a plug-in, the *DtDe* is passed to the plug-in's initDeModule() function.
- In a custom application, you ask the main application framework class (***DtVrvApplication*** for a pointer to its *DtDe*. (For details, please see "Building a New VR-Vantage-Based Application," on page 2-6.)

Switching between the two approaches is usually quite straightforward, if your new code is really an application extension.

2.3. Embedding VR-Vantage into a Larger Application

Some VR-Vantage Toolkit users will want to embed visualization capabilities into a larger (perhaps pre-existing) application, such as a Debrief or After-Action-Review system, an Instructor/Operator station, Command and Control System, or Mission Planning Tool. If this is your goal, then you will not develop plug-ins or modify the user*application* application frameworks. Instead, you will probably instantiate a *DtDe* directly, and call its tick() function at appropriate times within your application.

You will need to reparent the VR-Vantage rendering window so that it fits into your own application's GUI, and you will have to reconcile our event-handling mechanism with that of the larger application. If your application's GUI is based on Qt, our toolkit provides extra assistance in these areas. If you are using Microsoft Foundation Classes, or another GUI toolkit, additional work is required. (The **exampleMFC** example shows how to integrate VR-Vantage into an MFC applications.)

2.4. Getting Started with the VR-Vantage Toolkit

This section briefly explains how to register a plug-in and how to build a basic VR-Vantage application.

2.4.1. Creating a Plug-in

A typical VR-Vantage application (such as VR-Vantage Stealth) creates a *DtDe*, then immediately creates a ***DtPluginManager*** – a class responsible for loading and initializing plug-ins. The Plug-in Manager loads plug-ins before DtDe is initialized, because a plug-in developer often wants to affect the way a *DtDe* is initialized. The *DtPluginManager* loads each of the plug-ins in the *./plugins* folder, and tries to call a global function called initDeModule(). The Plug-in Manager passes a pointer to the *DtDe* to initDeModule(), so that the plug-in has access to the *DtDe*. Therefore, the main responsibility of every plug-in developer is to implement and export (with C linkage) a global function called initDeModule().

What you actually do in your plug-in is up to you. A typical plug-in may subclass one of the key classes and register the subclass with the display engine within initDeModule(). For example, the following plug-in code registers a new subclass of ***DtChannel*** (the class that represents visual channels) with the display engine:

```
// This method is called by the DtPluginManager during startup.
void initDeModule(makVrv::DtDe* de)
{
  // Tell the display engine to use an instance of MyChannelCreator
  // instead of the base DtChannelCreator to create channels.
  // MyChannelCreator will create MyChannels instead of base
  // DtChannels.
  DtChannelCreator::registerInstance(de, new MyChannelCreator);
}
```

Once you build your plug-in, place it in the *./plugins* folder, so that *DtPluginManager* can find it. The **exampleChannelPlugin** example shows how to create a simple plug-in.

Scheduling Actions after a Plug-in is Loaded

If you want a particular action to take place after a plug-in has been loaded, you can do so. For two examples of this, see **exampleAccesory** and **exampleCustom**.

2.4.2. Building a New VR-Vantage-Based Application

While plug-ins provide a flexible way to deploy extensions to VR-Vantage applications, there may be cases in which you need to create new applications based on our application framework. When you do this, we recommend that you use one of the user*application* examples as a starting point. The userStealth application source code looks like this:

```
int main(int argc,char** argv)
{
  // Show the splash screen.
  makVrv::DtVrvApplication::ShowSplashScreen(
    "../data/Overlays/SplashScreen_VR-Vantage_Stealth.png");

  // Create the Application.
  makVrv::DtStealthConfiguration config;
  makVrv::DtVrvApplication stealthApp(config);
  makVrv::DtStealthCommandLineProcessor cmdLineProcessor;
  stealthApp.installCommandLineProcessor(&cmdLineProcessor);

  // Parse Command Line.
  stealthApp.initialize(argc,argv);

  // Run.
  stealthApp.run();
}
```

Notice that we did not directly need to create a *DtDe*. That is because the higher-level class *DtVrvApplication* creates one for us. It also creates a *DtPluginManager* and loads any available plug-ins, and then initializes the *DtDe*. It sets up an event loop, and once you call run(), it repeatedly calls DtDe::tick(). To get access to the application's *DtDe*, simply call DtVrvApplication::de().

Please see the **exampleCreator** example to see how to derive a VR-Vantage application that lets you add some post initialization functionality.

2.4.3. Embedding VR-Vantage into an Application

The way you embed VR-Vantage functionality into a larger application depends on what the larger application looks like – how it is built, what GUI toolkit it uses, how you want it to interact with the VR-Vantage functionality, and so on. The **exampleEmbedded** example shows how to embed VR-Vantage functionality in an application.

2.4.4. Examples

VR-Vantage includes many examples of plug-ins and VR-Vantage applications. They are listed and documented in the Examples section of the class documentation.

2.5. Architectural Overview

This section provides an overview of the VR-Vantage Toolkit architecture. It describes the main classes and how they relate to each other, and provides some insight into which classes you will probably have to work with to accomplish particular development goals.

To understand the VR-Vantage Toolkit, you need to understand that it is a multi-layered toolkit, as follows

- OpenGL layer. The lowest layer is OpenGL, the standard low-level API supported by most graphics cards.
- OpenSceneGraph layer. OpenSceneGraph (OSG) is an open-source C++ toolkit that helps organize 3D geometry into a "scene graph", and provides functions for traversing the graph and making the appropriate calls to OpenGL to render the scene. It provides data management, LOD management, file loaders, and other features. For complete information about OSG, please go to http://www.openscenegraph.org/projects/osg
- VR-Vantage Rendering System layer. The classes at this layer associate some semantics with the geometry. Instead of just thinking of 3D geometry as generic "nodes", the Toolkit supports application-level concepts like terrain, props, the environment, models, and scene objects. The rendering code is independent of OSG. Theoretically, you could swap out OSG for some other software if needed. The rendering layer (*DtRenderer*) has an update() and render() method that it calls every tick.
- VR-Vantage Control Interface. This layer provides agents, proxies, and higher-level objects that allow you to control objects in the Rendering System independently of whether they exist in your application or in a remote display engine. (This allows VR-Vantage applications to support multi-channel distributed-rendering).
- Driver layer. The driver layer is where your application logic goes. Drivers get information from the networked simulation or user and tell the Rendering System (through the Control Interface) what objects to create, where to place them, and how to display them.
- VR-Vantage GUI layer. The GUI layer provides the graphical user interface for the application, including menus, toolbars, dialog boxes, and so on. It is based on the Qt Toolkit.
- Application layer. The application layer is a framework for a VR-Vantage application. The DtVrvApplication class reduces application development time by providing the basic building blocks of a VR-Vantage application, such as the display engine (*DtDe*) and Plug-in Manager (*DtPluginManager*).

Figure 2-1 illustrates the various layers of the VR-Vantage Toolkit.

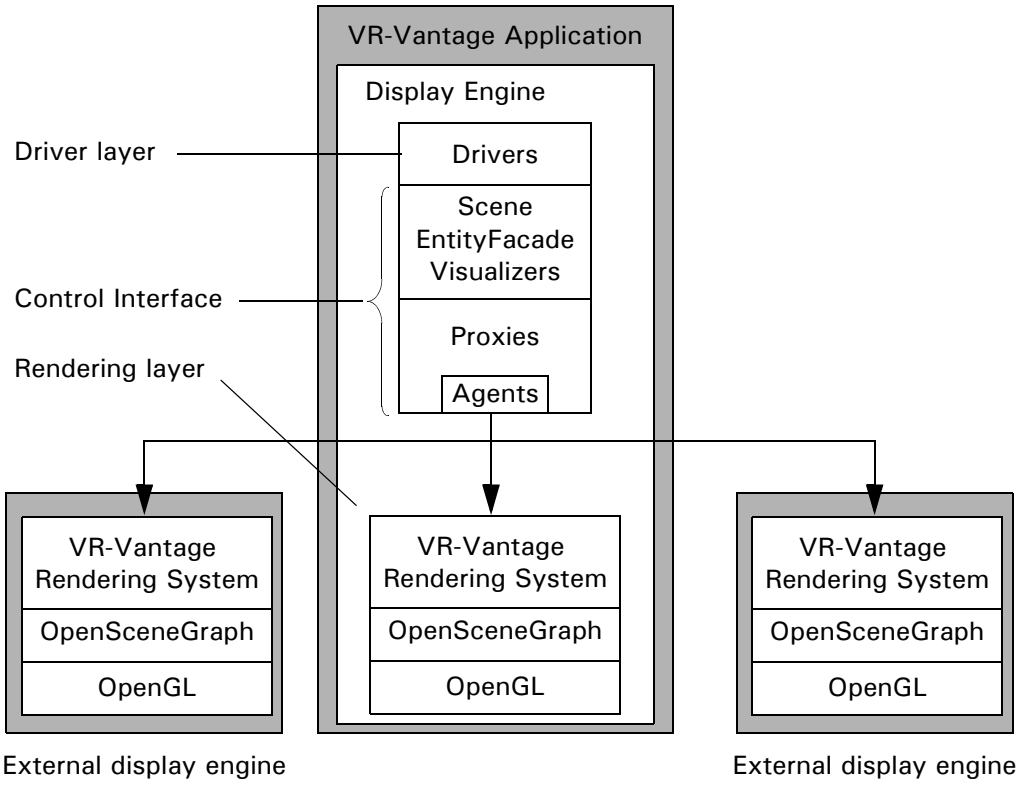


Figure 2-1. VR-Vantage application architecture

2.5.1. The Application Control Flow

This section looks at the control flow from top to bottom.

Drivers decide what needs to get drawn based on application business logic. For example, a DIS or HLA driver puts objects in the scene based on information it gets from remote simulation applications over the network; an input driver controls the location of the eyepoint based on user input.

Drivers communicate with both local and remote objects in the VR-Vantage Rendering System through the Control Interface (agents, proxies and other higher-level Control Interface objects). The Rendering System objects act on the commands that they receive from the Control Interface by making changes to the OSG Scene Graph, or by changing other OSG rendering parameters. OSG makes calls to OpenGL, which ensures that the correct scene is rendered by the graphics card.

DtDe, the top-level class, provides access to objects in all of these layers (as well as direct access to OpenSceneGraph!) You can write your application code so that it makes direct calls to the Rendering System (or to OSG). But if you just want to use the built-in capabilities of the VR-Vantage Toolkit (rather than trying to extend the capabilities of the toolkit), we recommend that you stay at the level of the Control Interface rather than going directly to the Rendering System. There are two reasons for this:

- The Control Interface layer provides more power. For example, a Control Interface class like ***DtEntityFacade*** allows you to associate multiple Rendering System objects with a simulation entity (for example, a vehicle) and control them all through a single interface. When you position an entity through a *DtEntityFacade*, not only will the entity's primary 3D model move through space, but trailing effects (vapor trails, and so on), track history graphics, and smoke and flames (if appropriate) will automatically follow.
- Your application will automatically support distributed-rendering. You or your application's users will be able to add additional display channels to your application merely by running the VR-Vantage Display Engine application, and connecting to it from your application.

If you want to extend the capabilities of the VR-Vantage Toolkit, you will need to work at the level of the Rendering System. For example, if you want to incorporate a new OSG nodekit into VR-Vantage, or if you want to generate geometry procedurally rather than loading and positioning existing 3D models or effects, you will need to subclass Rendering System objects. (You will also have to provide agents if you want applications working at the Control Interface level to be able to control your new objects.)

2.5.2. Roadmap to VR-Vantage Toolkit Classes

The VR-Vantage Toolkit does not group all of the Rendering System classes under a single "Rendering System" object, and all of the Control Interface objects under a "Control Interface" object. Instead, both sets of classes are accessible from *DtDe*. So it can sometimes be difficult to tell which classes are parts of which layer. For the most part, classes that end in the word "Object" represent scene objects that are part of the Rendering System. Classes that end in the word "Agent" are the agents that are used in the Control Interface to pass commands to their corresponding scene objects. There is usually a one-to-one relationship between scene objects and the agents that command them. For example, a *DtObserverObjectAgent.hpp* is used to control ***DtObserverObjects*** in the Rendering System.

Generated classes have the extension *.hpp* or *.cpp*.

However, in many cases, a higher-level class in the Control Interface layer groups together several agents, and provides an interface that allows you to control several different Rendering System objects from a single object (through multiple agents). Two important examples of this are ***DtScene*** and ***DtEntityFacade***. They use a variety of agents to control several objects related to terrain, the environment, and moving entity models.

The **exampleDriver** and exampleDriverPlugin examples use *DtScene* and *DtEntityFacade* to load a terrain database and place a 3D model of an F16 into the world.

In general, when you create a particular Control Interface object (for example, *DtEntityFacade* or one of the agents), the corresponding Rendering System objects will be automatically created in all display engines to which you are currently connected (including the one that is local to your application). The only thing that the Control Interface objects need to communicate with the Rendering System is a pointer to an interface object called an Agent Manager. The Agent Manager is created automatically by a *DtDe*, and is accessible by calling displayEngine->agentManager().

The next few sections briefly describe some of the key classes in the Rendering System layer and the Control Interface layer. Figure 2-2 illustrates the class relationships in the two layers as described in these sections.

2.5.3. Displays, Windows, and Channels

A *DtDe's* ***DtDisplay*** is the Rendering System layer object that manages the display engine's system of windows and channels. (Each display engine can own one or more windows, and can be configured to render one or more 2D or 3D channels into each window. As in most visual systems, a channel is a rendering of the scene from a particular point of view). You can get to a *DtDe's DtDisplay* by calling displayEngine->display().

The *DtDisplay* owns a ***DtWindowManager***, which manages a list of *DtWindows*. Each ***DtWindow***, owns a ***DtChannelManager***, which manages the list of ***DtChannels*** that are associated with a particular window. However, all of these classes are part of the Rendering System layer, which means that if you use these classes, you will only be able to access and control windows and channels that are owned and rendered by the display engine that is local to your application.

To control displays, windows, and channels independently of whether they are owned locally or by an external display engine application, use the corresponding Control Interface classes: ***DtDisplayProxy***, ***DtWindowProxy***, and ***DtChannelProxy***. Get a pointer to the *DtDisplayProxy* object associated with each display using *DtDe's* findDisplay() or findDisplayByIndex() functions. Once you have the *DtDisplayProxy*, you can use it to get pointers to window and channel proxies. For more detail about displays, windows, and channels, please see "Channels, Windows, and Displays," on page 4-3. For information about proxies, please see "Proxies," on page 4-6.

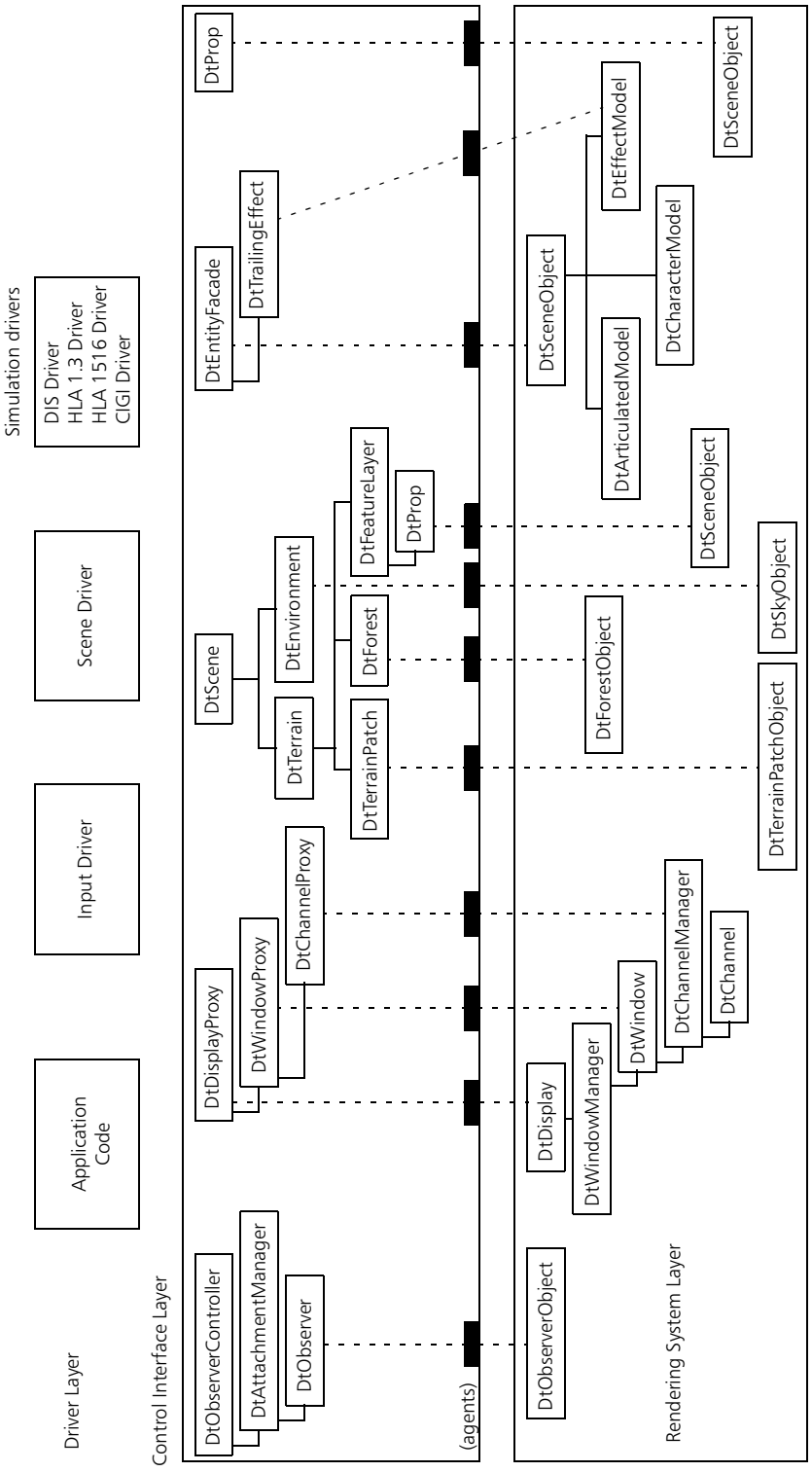


Figure 2-2. Class roadmap

2.5.4. Terrain and Environment

The Control Interface class *DtScene* manages the terrain and environment. It delegates much of its work to the ***DtTerrain*** and ***DtEnvironment*** classes. *DtTerrain* further delegates to:

- ***DtTerrainPatch*** to manage each "patch" of terrain
- ***DtFeatureLayer*** to manage layers of point features that are overlaid on top of the terrain
- ***DtForest*** to manage trees.

All of these classes exist in the Control Interface layer. They command the Rendering System classes ***DtSkyObject***, ***DtTerrainPatchObject***, and ***DtForestObject*** through their respective agents: ***DtSkyObjectAgent***, ***DtTerrainPatchObjectAgent***, and ***DtForestObjectAgent***. For details about agents, please see "Distributed Objects and Agents," on page 4-7.

The *DtDe* will automatically create a *DtScene* for you when it is initialized. Use displayEngine->scene() to get a pointer to the *DtScene*, and use its members to access and manipulate the *DtTerrain* and *DtEnvironment*.

2.5.5. Entities and Props

The main components of any 3D scene are the terrain and environment discussed in previous sections, and entities and props. Props are static objects such as buildings or Jersey barriers (like the props that are placed on a stage for a play). Entities are moving objects such as vehicles, missiles, and human characters (the actors in the play).

DtEntityFacade is the main Control Interface level class used to add an entity to the scene and to control its location and state. It uses another Control Interface level class called ***DtTrailEffect*** to manage attached effects. *DtEntityFacade* commands Rendering System classes such as ***DtSceneObject***, ***DtArticulatedModel***, and *DtEffectModel* through their respective agents. As shown in the example in "Roadmap to VR-Vantage Toolkit Classes," on page 2-10, you can create an instance of *DtEntityFacade* for each entity that you want to add to the scene, then call its member functions (or members of objects that are owned by *DtEntityFacade*) to position the object and set additional state.

The ***DtProp*** class is used to create and manipulate props. Like *DtEntityFacade*, you can create an instance of *DtProp* for each prop you want to add to the scene (although it often makes sense to let a *DtTerrain*'s *DtFeatureLayer* manage *DtProps* for you). *DtProps* command Rendering System classes such as *DtSceneObject* and ***DtModel*** through their respective agents.

2.5.6. Observers and Observer Attachment

DtObserver is a Control Interface level class that represents an observer – that is, an eyepoint into the virtual world. Through a *DtObserver*, you can control the location of the corresponding ***DtObserverObject*** that exists in the Rendering System. In order to associate a *DtObserver* with a channel, just make sure the *DtObserver*'s name matches the observer name specified by your channel's configuration:

```
// Channel config initialized elsewhere.
DtChannelConfiguration channelConfig = getChannelConfigFromSomewhere();
DtObserverConfiguration configuration(channelConfig.observerName());
DtObserver* observer = new DtObserver(de, de.agentManager(),
  configuration);
```

You can call *DtObserver*'s member functions directly to set observer location and other state. However, we recommend using one of two higher-level classes to call *DtObserver* functions on your behalf: ***DtAttachManager*** or ***DtObserverController***. Use *DtAttachManager* to perform attachment-related operations, so that other parts of the VR-Vantage system are notified when attachment changes. *DtObserverController* implements a higher-level navigation interface on top of *DtObserver*. It lets you perform operations like "move left" and "pitch up", calculating resulting changes to the observer and calling *DtObserver* functions for you.

2.5.7. Drivers

In the previous sections, we described how the control logic of your application can use Control Interface classes to control objects in the Rendering System. Control Interface classes like *DtEntityFacade* and *DtScene* do not "decide" what to draw – the code that uses those classes does. In the example in "Roadmap to VR-Vantage Toolkit Classes," on page 2-10, it was the application code itself that decided that the application should load *makland_fast.mtf*, and that it should add an F16 model to the scene. As that example showed, the Toolkit does not require you to put your control logic in any particular place. You can instantiate classes like *DtEntityFacade*, or access the *DtDe*'s *DtScene* object from anywhere in your application – in main(), in your GUI code, and so on. However, we recommend that you organize your control logic into objects called drivers.

Application logic is in the drivers. Drivers are implemented as subclasses of the ***DtDriver*** class. If the Rendering System objects are like the actors and props on a stage, then a driver is like the director of the play. Drivers direct the action by communicating their intent to the actors (Rendering System objects) through their agents (or other Control Interface classes).

Putting your control logic into VR-Vantage drivers has the following benefits:

- Drivers are automatically "ticked" every time the *DtDe* is ticked, so that you do not have to worry about making sure your logic is executed at the right time.
- VR-Vantage lets you start and stop drivers through the GUI. This means you have run-time control over the execution of your control logic if you put it into a driver.
- *DtDrivers* automatically find out when external display engines connect or disconnect from your application and provide a place to put logic that needs to be executed when those events occur. In other words, since you are the one deciding what goes in the scene, it is your responsibility to ensure that late-joining display engines find out what you have already put in the scene before it joined. And the *DtDriver* class helps you to meet that responsibility,

The VR-Vantage applications have plenty of application-level logic built in. *DtDrivers* hold that logic, for example:

- The ***DtInputDriver*** is responsible for processing mouse and keyboard input, and making calls to objects like the *DtObserverController* or *DtAttachManager* based on the control input received from the user.
- The ***DtDisDriver***, ***DtHla13Driver***, and ***DtHla1516Driver*** "drive" the contents of the scene based on information received from the network through their respective simulation protocols.
- The ***DtSceneDriver*** makes sure that a new scene is created when a *DtDe* is initialized, and can populate a scene based on information in a saved scene file.

If your application is a flight simulator with an integrated visual/simulation configuration, you might create a "SimulationDriver" that drives a *DtEntityFacade* and the eyepoint based on the state of your ownship simulation object as it is computed each frame.

If you want to add support for a new simulation networking protocol to VR-Vantage, you would probably create a new kind of *DtDriver* for your protocol. And if you wanted to extend the set of DIS PDUs or HLA FOM attributes that cause updates to the scene, you would probably derive from our existing protocol drivers. The **exampleFomMapper** example shows how to extend drivers to listen for attributes in a modified FOM.

For more information about drivers, please see Chapter 5, *Drivers*.

GUI Architecture

This chapter describes the VR-Vantage GUI architecture and various runtime implementation details.

3.1. GUI Architecture

All of the VR-Vantage GUI elements (menus, menu items, and dockable widgets) and dialog box pages and pagers are provided by plug-ins. The presence of the GUI in applications based on the *DtVrvApplication* class is controlled by its configuration's assembleUserInterfaceComponents() member. If you want to run MÄK's VR-Vantage applications without the GUI, you can use the `--noGui` command-line option to set this member in the ***DtVrvApplicationConfiguration***.

The GUI is constructed by the ***DtQtMenuAssembler***, the ***DtQtToolbarAssembler***, and the ***DtQtPageAssembler***. To add or remove GUI elements, register or unregister the appropriate builder with the appropriate assembler.

3.1.1. Menus

VR-Vantage's menu system is configured by each application, controlled by the vrvCore module, and implemented in various plug-ins. Menus are derived from ***DtMenu***.

The implementation for the menu system is part of the Qt plugin (*vrvCoreQt.dll*). This plugin defines concrete menus and menu items with Qt-based implementations, such as ***DtAddFeatureLayerItem.h***, and registers their creators with the *DtDe*. (For details, please see ***DtMainMenus.h*** and the various classes implementing ***DtMenuItem***. Each menu and menu item has a unique ID name provided by its constructor, such as DtExamplePage, that is used to identify it.

The menu interface used directly by the *DtDe* is defined in the vrvCore module by the classes ***DtMenuAssembler***, ***DtMenuConfiguration***, and ***DtMenuPath***. The *DtMenuPath* class is used to specify the location of a single menu or menu item in terms of its location and the name of the creator that generates its implementation.

DtMenuConfiguration contains a collection of menu paths and provides several methods to conveniently add, access, and modify menu paths. The *DtMenuAssembler* acts as a sort of menu configuration interpreter. It takes a *DtMenuConfiguration* and executes the creators necessary to create the menus and menu items. The Qt plugin overrides the default menu assembler to assemble the menus in a Qt-specific fashion. If you want to replace Qt with another GUI toolkit, you will probably need to implement your own *DtMenuAssembler* as well.

The *DtDeApplication* framework class has the functions masterModeMenuConfiguration() and slaveModeMenuConfig(). It uses them to generate the correct menu configuration. The source for MÄK's VR-Vantage applicationsVR-Forces overrides these functions to produce their respective menus. For details, please see the **exampleDialogPage** example.

3.1.2. Toolbars

Toolbars are assembled similarly to menus. Registered menu item creators can also be used to populate toolbars, and can be specified in toolbar paths. Toolbars can also contain objects derived from *DtWidget*. Creators for the widgets are registered with the ***DtQtToolbarAssembler***.

3.1.3. Dialog Box Pages

Most dialog boxes in VR-Vantage are implemented as pages in a tabbed parent dialog box, called a page collection. Each page derives from ***DtPage***, which is a QWidget. A page is installed into the system by making a creator class for it and registering that creator with the page assembler (please see **exampleDialogPage** for an example). Once registered, pages can be added to the GUI using the VR-Vantage application's page configuration.

3.1.4. Page Collections

Tabbed dialog boxes that display pages are called page collections. The set of page collections that exists at runtime is determined by the VR-Vantage application's page configuration. (Please see the exampleDialogPage example for an example of adding a new page collection.) Page collections can be configured as either dockable or free-floating dialog boxes.

3.1.5. Standalone Dialog Boxes

A few dialog boxes in VR-Vantage are standalone dialog boxes, rather than pagers or pages. These cannot be registered with the user interface assemblers, so they should be avoided if possible.

3.2. The Event Loop

The *DtDe* needs to be ticked regularly by an event loop. We provide a default event loop, ***DtEventLoop***, which you can customize by passing it a function pointer to your own event loop function. Use this method for applications that do not use the VR-Vantage Qt plug-ins.

3.2.1. Qt Event Loops

Qt requires you to use a Qt event loop. Therefore, if you are using one of the Qt plug-ins, it is impossible to use the *DtEventLoop* as your main event loop. Since the ***DtVrvApplication*** calls whichever *DtEventLoop* is registered with the *DtDe*, in order to use Qt and *DtVrvApplication* together you must create a *DtEventLoop* class that calls QApplication::exec() in its run function, and also make sure that the Qt event loop ticks the *DtDe* regularly, for instance with a QTimer. The ***DtQtEventLoop*** class installed by the vrvQtWidgets library does this with ***DtQtDeTicker***. You can implement your own Qt event loop by subclassing QEventLoop. Please see the Qt documentation for more information about how to do this.

3.3. Runtime Settings

The VR-Vantage applications have various runtime settings, for example, enabling and disabling entity labels. These settings are owned by Settings Managers. Settings at runtime are ***DtDeSettings*** objects, owned by the ***DtDeSettingsManager***. They are saved into the *./appData/settings* directory, and the factory defaults are in *./factory/settings*.

3.3.1. Settings Directory Structure

The *./appData/settings* directory is the root directory for user settings, and each application has its own subdirectory (for example, *./appData/settings/stealth* and *./appData/settings/vantageIG*). Similarly, *./factory/settings* is the factory settings root directory, with subdirectories for each application. The settings directories are configured by the ***DtDePathConfiguration***. When an application wants to load a default settings file, it should check the user settings directory first. If no appropriate file is found, the factory settings directory should be checked.

3.3.2. Loading and Saving Settings

Each *DtDeSettings* object is responsible for loading and saving itself. The settings objects can load and save from default files, factory files, or a specified file path. Please see ***DtDeSettings.h*** for detailed Toolkit documentation. You can get the *DtDeSettings* object for a particular group of settings from the *DtDeSettingsManager*, as follows:

```
DtDeSettings* settings =
  DtDeSettingsManager::instance(myDe).findSettings("Settings Name");
```

3.4. Records

VR-Vantage uses Boost's serialization functionality for reading and writing files. A class that contains data that is to be read from disk and written to disk is called a record.

Record classes are part of the makArchives library. The guidelines for classes in this library are as follows:

- All classes should be Records, and constructed of POD/Value semantics.
- All classes should be designed for efficient file storage.
- All files saved by VR-Vantage must represent a single Record that can be loaded using a single call.

Basic usage of records is as follows:

```
const std::string filename = "name.ext" SomeDataRecord rec;
//To read a file.
makArchives::DtXmlFileIo::read(filename, rec, "Name");
//To write a file.
makArchives::DtXmlFileIo::read(filename, rec, "Name");
```

For the most part there is a specific DataRecord type for each file type. However, some records deviate from this standard. All files in *./appData/drivers* are ***DtDriverRecord*** files, for example, *DIS.DtDisDriver* stores a *DtDriverRecord*. The file extension (*.DtDis-Driver*) is the name of the driver class needed to use the file data.

Here is an example record:

```
class DtTestRecord
{

  public:

    // All values in a record  *MUST* be initialized.
    DtTestRecord()
      : degrees(0)
      , minutes(0)
      , seconds(0.0)
    {
    }

    DtTestRecord(int d, int m, float s)
      : degrees(d)
      , minutes(m)
      , seconds(s)
    {
    }

    int degrees;
    int minutes;
    float seconds;

  protected:

    friend class boost::serialization::access;

    // When the class Archive corresponds to an output archive, the
    // & operator is defined similar to <<. Likewise, when the class
```

```
        // Archive is a type of input archive the & operator is defined
        // similar to >>.
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version)
        {
          ar & BOOST_SERIALIZATION_NVP(degrees);
          ar & BOOST_SERIALIZATION_NVP(minutes);
          ar & BOOST_SERIALIZATION_NVP(seconds);
        }
    };
```

To write a test record:

```
std::string filePath = "testRecord.xml";
DtTestRecord data(35, 59, 24.567f);
makArchives::DtXmlFileIo::write(filePath, data, "DtTestRecord");
```

To read a test record:

```
std::string filePath = "testRecord.xml";
DtTestRecord data;
makArchives::DtXmlFileIo::read(filePath, data, "DtTestRecord");
```

3.5. Configuration Files

VR-Vantage saves most configuration files as XML text files. Most files represent a single MAKArchive record encoded as XML. Each file generated by a different record class has a different file extension.

Table 3-1 lists the file extension, the record used to read and write the data, and the contents of the various configuration files saved by VR-Vantage.

Table 3-1: VR-Vantage file formats

Extension	Record Class	Description
adsx	*DtAggregateSettingsRecord*	Aggregate display settings.
dcx	None	A display engine configuration.
DtDisDriver DtHla13Driver DtHla1516Driver	*DtDriverRecord*	An instance of a driver to be created on startup.
edsx	*DtSharedSimulationSettingsRecord*	The simulation display settings.
elem	*DtElementDefinitionRecord*	Element definitions.
envx	*DtEnvironmentRecord*	The environment settings.
forx	*DtForestSettingsRecord*	The forest drawing settings.
grdx	*DtGridLinesSettingsRecord*	Grid line settings.

Table 3-1: VR-Vantage file formats

Extension	Record Class	Description
invx	*DtIntervisibilitySettingsRecord*	Intervisibility settings.
kmpx	*DtKeyMapRecord*	The keyboard to function mappings.
magx		Aggregate mappings.
mdtx	*DtEntityTypeMappingRecord*	The entity type mappings for detonation effects.
metx	*DtEntityTypeMappingRecord*	The entity type mappings for entity models.
misx	*DtInterestManagementRecord*	Interest management settings.
mlsx	*DtLoaderSettingsRecord*	Loader settings.
mrsx	None	The render settings.
mtgx		Tactical graphics mappings.
mwfx	*DtEntityTypeMappingRecord*	The entity type mappings for weapon fire effects.
omcx	*DtObserverSettingsRecord*	Observer mode settings.
ommx	*DtModelDefinitionRecord*	Model definitions.
osrx	*DtObserverStateRecord*	Observer state record (saved views file).
smtx		Schema settings.
symx	*DtSymbolDecorationSettingsRecord*	Symbol decorator settings.
tdsx	*DtTacticalGraphicsSettingsRecord*	Tactical graphics display settings.
uisx		GUI layout settings.
usdx	*DtCacheSettingsRecord*	The caching settings for the application.

The Display Engine

This chapter describes the role of the display engine, display management, and distributed rendering.

4.1. The Display Engine

A VR-Vantage application can be defined as any application that has a display engine in it. A display engine contains almost all of the major systems and subsystems that are described in this manual. Among the systems that the display engine owns are:

- Display Engine Initializer (*DtDeInitializer* – Contains the data structure and parameters the display engine was initialized with.
- Display Engine Communicator (*DtDeCommunicator* – The component that handles communication between master and external display engines. It uses TCP and handles the lower level mechanics of distributing messages. Proxies use the communicator directly.
- Display (*DtDisplay* – Manages windows and channels. It owns the display configuration for one display engine - the Window Manager and Channel Manager. The master display engine has proxies for all of the other display engines.
- Scene (*DtScene* – The object that represents everything in a scene (terrain and environment). The scene signaler signals scene-level changes.
- Simulation Data (*DtSimData* – The repository of simulation-specific, read-only data.
- Driver Manager (*DtDriverManager* – The display engine's container of drivers.
- Display Engine Shared State (*DtDeSharedState* – Owns all data shared between master and external display engines
- Renderer (*DtRenderer* – Draws 3D images onto a screen. It owns the scene rendering root objects, the scene graph and overlay managers.
- Agent Manager (*DtAgentManager* – manages the distributed scene object framework. This includes the objects and updates as well as the drivers used to control the objects.
- Data Bank (*DtDataBank* – The Data Bank is the repository for common element data and simulation-specific state data.

4.1.1. Channels, Windows, and Displays

A display (*DtDisplay*) owns and manages windows. A display can own zero or more windows. Windows can own zero or more channels.

To see a rendered scene, you need a display configured with at least one channel inside a window. A channel is a area on the screen onto which a camera's view of the scene is rendered. Channels live inside windows. VR-Vantage has the following types of windows:

- Fullscreen window - A window that takes up the whole screen with no surrounding GUI frame or elements.
- Embedded window - A GUI toolkit-specific element inside an application (that is, a Qt Widget).
- Window - A resizable window with a frame and the windowing system controls (maximize, close, and so on).

Creating a Display Configuration

You specify the number and organization of displays, windows and channels in a display engine by creating a display configuration. You can create a display configuration programmatically and then have it created by a display engine.

Display configurations must have unique names. The display engine will not implement configurations with duplicate names. For an example of creating a display configuration programmatically, please see the **exampleConfiguration** example in the class documentation.

4.1.2. Channel Keywords

Channel keyword objects are objects that are created due to the presence of a keyword in a channel's configuration. Possible types of channel keyword objects might include compasses, logos, cockpit displays, message logs, and so on. When a channel gets created, any objects specified by the associated keywords get created, such as a compass, or can be created, such as a cockpit display. If the keyword is removed from a channel by a user, the object gets removed from the scene.

Channel keyword objects are managed by the *DtChannelKeywordObject*, *DtChannelKeywordObjectFactory*, and *DtChannelKeywordObjectManager* classes.

DtChannelKeywordObject

DtChannelKeywordObject is an abstract class that provides a common interface for channel keyword objects.

DtChannelKeywordObjects are created in a factory using *DtChannelKeywordObjectCreators*. ***DtChannelKeywordObjectCreator*** has a create() method that takes the same arguments as *DtChannelKeywordObject*, except for the keyword parameter. The creator class itself provides a keyword to the *DtChannelKeywordObject* constructor via a static keyword() method. In this way, the keywords are tied to the creator objects, rather than just being "magic strings" that appear in the system. For details about this class's methods, please see the class documentation.

DtChannelKeywordObjectFactory

The *DtChannelKeywordObjectFactory* is a singleton class that allows you to register channel keyword object creators, and which is used by the *DtChannelKeywordObjectManager* to create channel keyword objects. For details about this class's methods, please see the class documentation.

DtChannelKeywordObjectManager

The *DtChannelKeywordObjectManager* is a singleton object with an instance() accessor. The instance() accessor uses a registered creator to create the manager (registered in *vrvCore.cxx*). *DtDe* instantiates the class after initialization (and before postInitialize) so that it exists before any channels are realized.

The manager assumes it will exist by the time the first channel is created – it does not search for existing channels when it is constructed.

The *DtChannelKeywordObjectManager* listens to the display engine's signal_channelConfigurationModified, signal_channelToBeDestroyed, and signal_windowConfigurationModified as well as the DtChannelManagerSignaler's signal_channelCreated signals to manage the display of channel keyword objects within VR-Vantage.

When a channel is added, *DtChannelKeywordObjectManager* looks for keywords in the configuration for which there are entries in the *DtChannelKeywordObjectFactory* and if they exist, creates a channel keyword object for that channel.

When a channel configuration is modified, *DtChannelKeywordObjectManager* determines if the channel needs a channel keyword object added or removed. If channel keyword objects already exist for that channel, the objects will be notified via their channelConfigurationModified() methods. If an object is to be removed, *DtChannelKeywordObjectManager* calls the object's objectToBeDestroyed()method and then deletes or removes it. Additionally, it checks to see if the channel name has changed. If it has, it adjusts the manager's maps for looking up the channel keyword objects on a channel by display/window/channel name.

When a channel is destroyed, *DtChannelKeywordObjectManager* determines if it has any channel keyword objects for that for that channel and if so, destroy them.

When a window configuration changes, *DtChannelKeywordObjectManager* notifies any channel keyword objects in that window by calling their windowConfiguration-Changed() methods. It also checks to see if the window's name has changed. If it has, it adjusts the manager's maps for looking up the channel keyword objects on a channel by display/window/channel name.

DtChannelKeywordObjectManager maintains an std::map of channelName :: windowName :: displayName to keyword / *DtChannelKeywordObject* pairs for each channel keyword object created. Given a channelName :: windowName :: display-Name, the map provides a list of *DtChannelKeywordObjects*, one for each keyword that exists. It will also maintain a map of displayName::windowName to a list of channel-Name :: windowName :: displayName strings. This map makes it possible to get all the channel keys for channels that have channel keyword objects on them, given a window key.

4.2. Distributed Rendering

The VR-Vantage Toolkit, and applications built on it, support distributed rendering. Distributed rendering is the process of rendering a single scene simultaneously on two or more computers. Distributed rendering is managed by a Control Interface running in a display engine in a VR-Vantage application. The Control Interface takes input from drivers and manages its rendering system as well as those of external display engines.

In distributed rendering, the VR-Vantage application's display engine sends commands to its Rendering System and to connected display engines, which execute them. Since each external display engine executes the same commands, they each display the exact same scene as the master and the objects perform the same actions in the same order. Since the scenes are synchronized across all display engines, the rendering of the scene is also synchronized. However, the scenes are only loosely synchronized, as only the scene itself is synchronized, not the rendering of the scene. VR-Vantage does not support low-level rendering synchronization such as frame locking, which ensures that each channel renders at the same frame rate, or Genlock, which ensures that each monitor refreshes the image at the same time.

The commands sent by the display engine are really just function calls. The Toolkit makes it easy to make a function call to a single C++ class instance and have that call get made on one or more objects, existing on one or more display engines.

4.2.1. Proxies

The master display engine contains the complete state of all connected display engines. However the master does not have an actual copy of all objects on the display engines. Instead, it has proxy objects. For example, there is a *DtWindow* object for each window. To represent a window on a distributed display engine the master display engine has a *DtWindowProxy* object, which is a proxy representation of the real object. Calls made on a proxy object are distributed to the display engine that has the real object, which behaves as if the call had been made directly to it.

The communication from a proxy object to a real object is performed by an agent object. An agent instance exists for each proxy/class pair. The master display engine makes calls to an agent and the agent distributes those calls automatically.

4.2.2. Distributed Objects and Agents

Objects that exist in display engines are called distributed objects. Each object in the scene is composed of an agent object and a distributed object. When the master display engine creates an agent, each external display engine simultaneously creates the distributed object that the agent will control.

An agent is created with an appropriate subset of the methods that the actual object has. For example, if a distributed object has a setLocation() method, then its agent could have it. On the master, you would call a method on the agent, as follows:

```
agent->setLocation(x,y,z);
```

The distributed rendering architecture handles all the data marshalling (encoding, transport, and decoding) and the corresponding object on each display engine has the setLocation() method called.

Each display engine owns an Object Manager. This is a container of distributed objects. At any given time the contents of the Distributed Object Manager on the master display engine and on each of the external display engines is the same.

4.2.3. Distributed Rendering Transport

The VR-Vantage Toolkit provides mechanisms for implementing communications over any protocol or medium. The implementation provided in the Toolkit and used by MÄK's VR-Vantage applications uses a TCP socket connection. Once a display engine is connected, most rendering will be distributed unless the application is explicitly designed not to use the distributed rendering functionality.

4.2.4. Example of Using an Agent

The following pseudo code demonstrates use of agents. Suppose you want to add a traffic light to your scene. You want to be able to programmatically change its state from green to yellow to red.

1. Create the TrafficLightObject. This is the object that will get created in each of the display engines and will make changes to the scene graph.

```
class TrafficLightObject
{
  public:

  ...

  void setState(int state)
  {
    myState = state;
    changeState();
  }

  void changeState()
  {
    // change the geometry in the scene graph to reflect
    // the traffic lights new state.
  }

  int state() const
  {
    return myState;
  }

};
```

2. Generate an agent. (For details about generating agents, please see "Generating Code for Agents," on page 4-9.) The agent will have a similar API to your object.

```
class TrafficLightAgent : public DtDistributedObjectAgent
{
  public:

  ...

  void setState(int state);
};
```

3. Create an instance of the agent in a master display engine's driver:

```
TrafficLightAgent* agent = TrafficLightAgent::create();
```

   This causes TrafficLightObject's to get created on each display engine (the master display engine also) and added to the Distributed Object Manager.

4. You can now make calls on the agent to set its state and each object's setState() method will get called.

```
agent->setState(State_Red);
```

The distributed object framework guarantees that the calls will happen in the order in which you make them. For example, if you make the following calls in your driver:

```
trafficLightAgent1->setState(State_Red);
trafficLightAgent2->setState(State_Green);
```

Each display engine will execute them in order:

```
trafficLightObject1->setState(State_Red);
trafficLightObject2->setState(State_Green);
```

Some drivers are written to run asynchronously in a separate thread. These drivers each have their own interface for setting messages from agents to the implementation. These commands are executed in the order in which they are called, however they may be interspersed with commands generated in the main thread or in other threads. For example, the agent might call functions in the following order:

```
Thread A: fooAgent->doSomething();
Thread A: fooAgent->doSomethingElse();
Main Thread: barAgent->doSomething();
Main Thread: barAgent->doSomethingElse();
```

The functions might be executed in the following order:

```
bar->doSomething();
foo->doSomething();
foo->doSomethingElse();
bar->doSomethingElse();
```

4.3. Generating Code for Agents

Every object needs an agent to manage its creation on the master display engine and external display engines. Since the generic functioning of agents is the same regardless of the object, it is possible to automate the process of creating them. VR-Vantage includes a tool, the Distributed Code Generator, that generates source code for agent classes from object classes so that they can be easily integrated into drivers.

The Distributed Code Generator generates an agent class, an agent implementation class, a definition class, and a creator class. Most of the time, you will only need to know about the agent class. However, the following is a brief description of all of the generated classes:

- The agent class is an abstract class that allows the agent to be completely independent of the dependencies of the object class
- The agent implementation class implements the agent functions and must have access to the object class
- The definition class stores the data regarding the agent functions to ensure that they are properly forwarded
- The creator class is used by the drivers and distributed object manager to create these classes.

The **exampleDistributedObject** example has generated agent classes. Please see the exampleDistributedObject page in the class documentation for a description of the object class, generating agent classes, and adding the agent to a driver.

4.3.1. The Object Class Definition

The Distributed Code Generator generates code based on the information in an object class definition (OCD). An OCD contains all the information the Distributed Code Generator needs to generate the agent classes, such as the object class name, namespace, functions, and comments. The functions that the agent will have are called distributed functions, because these function calls will be distributed to objects in display engines.

You create an OCD in the Distributed Code Generator's graphical user interface. You save the OCD in an object class definition file (extension *.ocdx*).

Once you create the OCD, you can build the classes from the Distributed Code Generator's GUI, from the command-line, or from within Visual Studio.

4.3.2. Distributed Code Generator Limitations

The Distributed Code Generator has the following limitations:

- Only methods that return void are supported. The agents on the master display engine always send commands to the external display engines. The external display engines do not send anything back. Therefore, no methods on agents can ever return anything.
- Agents do not have member values. All they do is forward function calls to the actual object.

4.3.3. Starting the Distributed Code Generator

You can start the Distributed Code Generator from the command line. For details, please see "Distributed Code Generator Command-Line Arguments," on page 4-14. On Windows, you can also start it from the Start menu.

4.3.4. Creating an Object Class Definition

To see an example of an object class definition, load *./examples/exampleDistributedObject/DtExampleObject.ocdx*.

To create an object class definition:

1. On Windows, choose **Programs** → **MÄK Technologies** → **VR-Vantage 1.4** → **Tools** → **VR-Vantage Distributed Code Generator** or run *./bin/dcgen.exe*. On Linux, run *./bin/dcgen*. The Distributed Code Generator opens (Figure 4-1).

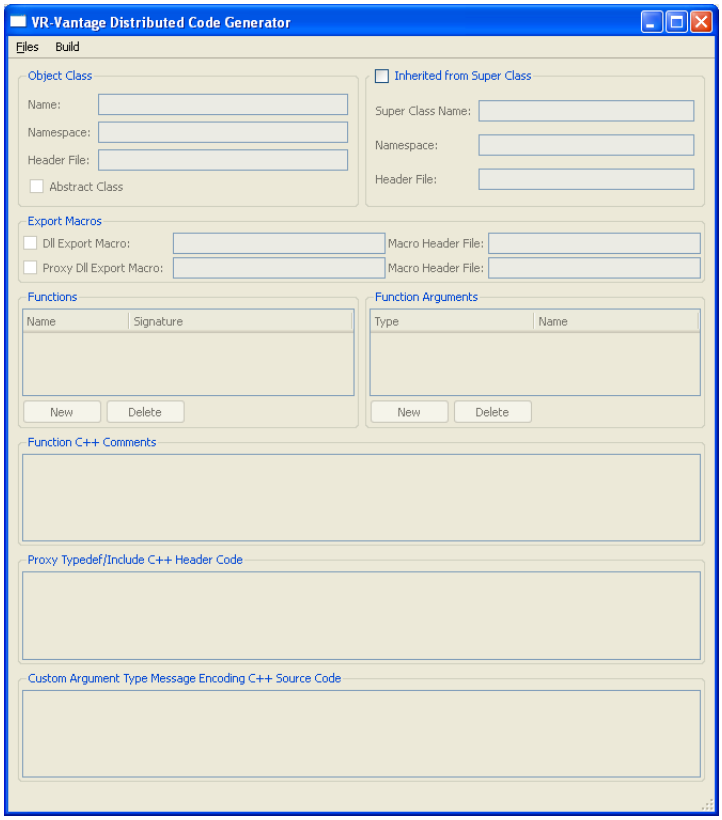


Figure 4-1. Distributed code generator

2. Choose **Files** → **New**.
3. In the Object Class group box, Name text box, type the name of the object class.
4. Type the namespace that the class is in.
5. Type the name of the header file for the object class.

6. Specify the agent functions. For each function:
   a. In the Functions group box, click New. The cursor jumps to the Name column and an empty signature is added.
   b. Type the function name in the name column.
   c. In the Function Arguments group box, click New. The cursor jumps to the Type column.
   d. Type the type of the argument.
   e. Double click the Name field. It becomes editable.
   f. Type the name of the argument.
   g. Type comments for the function in the Function C++ Comments text box.
7. Optionally, complete other fields in the form.
8. Choose **Files** → **Save**. The Save Class File dialog box opens.
9. Navigate to the location where you want to save the OCD file.

We recommend that you create a directory for your OCD files, for example, *myProject/Object Class Definitions*. Then create a subdirectory called *generated* to store the files generated by the Distributed Code Generator.

10. Type a name for the file.
11. Click Save.

4.3.5. Building Agent Classes

You can build agent classes from the Distributed Code Generator GUI, from the command-line, or as part of your build process using Microsoft Visual Studio. For command-line arguments, please see "Distributed Code Generator Command-Line Arguments," on page 4-14.

Building Agent Classes in the Distributed Code Generator GUI

To build agent classes from the Distributed Code Generator GUI:

1. Create an object class definition (as described in "Creating an Object Class Definition," on page 4-11) or load a previously created OCD.
2. Choose **Build** → **Set Output Directory**. A directory chooser dialog box opens.
3. Select the directory in which the Distributed Code Generator should put the generated source code.
4. Click OK.
5. Choose **Build** → **Generate Files**.

Building Agent Classes in Your IDE

You can integrate creation of agent classes into your build process. This is the recommended way to generate these classes.

This example code in this section procedure is from the **exampleDistributedObject** example. It assumes that you have created an OCD and saved it to *DtExampleObject.ocdx* and created a directory called *exampleDistributedObject*.

To generate the agent files using Microsoft Visual Studio:

1. Right click on the *.ocdx* file in the Solution Explorer and examine its Custom Build Step in its properties. It will look something like this:

```
Outputs:
$(InputDir)$(InputName)Agent.hpp;$(InputDir)$(InputName)Agent.cpp
$(InputDir)$(InputName)Classes.hpp;$(InputDir)$(InputName)Classes.cpp
Additional Dependencies: ..\..\bin\sceneObjectCompilerd.exe
Command Line: ..\..\bin\sceneObjectCompilerd.exe "$(InputPath)" -a -S
  "$(InputDir)/" -H "$(InputDir)/" -A "../../include"
```

2. Compile just the *DtExampleObject.ocdx* file (right click on it and click compile). This generates the following files in the exampleDistributedObject directory:
   - *DtExampleObjectClasses.cpp*
   - *DtExampleObjectClasses.hpp*
   - *DtExampleObjectAgent.cpp*
   - *DtExampleObjectAgent.hpp*.
3. Add those files to the *generated* folder.
   a. Right click on the *generated* folder and click add existing item.
   b. Select those files and add them.

4.3.6. Distributed Code Generator Command-Line Arguments

You can run the Distributed Code Generator from the command line if you have an object class definition file to input to it. The syntax is:

```
dcgend.exe input_file [{-i | -a | -p} -A root_dir -H
  header_dir -S source_dir -o output_dir -- -v -h]
```

Table 4-1: Distributed Code Generator command-line arguments

Argument	Description
input_file	The Object Class Definition file to process.
(-i | --generateImplementation)	Generate all of the implementation classes.
(-a | --generateAll)	Generate all of the classes.
(-p | --generateAgent)	Generate only the abstract driver class.
(-A | --RelativeIncludeDirectory) *root_dir*	The root directory that generated include statements are relative to.
(-H | --HeaderOutputDirectory) *header_dir*	The directory to write header files to.
(-S | --SourceOutputDirectory) *source_dir*	The directory to write source files to.
(-o | --OutputDirectory) *output_dir*	The directory to write all files to.
(-v | --version)	Display version information and exit.
(-h | --help)	Display usage information and exit.
(-- | --ignore_rest)	Ignore all command-line arguments after this one.

Suppose that you are generating the file *testAgent.hpp* in the */productRoot/include/test/* directory. However, you want the include statement to look like:

```
#include <test/testAgent.hpp>
```

and NOT

```
#include <testAgent.hpp>
```

Use the arguments `-H /productRoot/include/test -A /productRoot/include` to specify the header location and relative include path.

4.4. Observers

An observer is a special kind of distributed object that is not part of the scene. It controls the camera associated with one or more channels. The input driver is responsible for instantiating observers, because they provide the link between mouse and keyboard input and the cameras that determine how a scene is viewed. For details about how distributed objects are managed, please see "The Input Driver," on page 5-3.

Observers that control OSG cameras are implemented by the *DtOsgObserverAgent* and ***DtOsgObserverObjects***, which are derived from the *DtObserverAgent* and ***DtObserverObject*** classes respectively. The ***DtObserver*** class provides a renderer-independent interface for creating and controlling observer objects. The renderer creates and maintains an Observer Object Manager (***DtObserverObjectManager***).

The *DtObserverAgent* and *DtOsgObserverAgent* classes are generated. They do not have header files and are not in the class documentation.

When the input driver creates an observer object (through the *DtObserver* class), the observer is registered with the renderer's Observer Object Manager. Channels can then use the Observer Object Manager to look up the observers that have been designated as their controllers. The linkage between channels and observers is made by observer name. Each channel is given the name of the observer that it should use to position its camera. The observer name is specified in the channel configuration (default "Observer 1"). For details about creating observers, please see "The Input Driver," on page 5-3.

There are several Boost signals associated with observers. Signals that indicate the creation and destruction of observers are available using the ***DtInputDriverSignaler*** (since the input driver creates and owns all observers). You can also connect to the signals raised by individual observers. Table 4-2 describes these signals.

Table 4-2: Observer signals

Signal	Description
signal_objectAttached	Called when the observer attaches to an object. Includes the observer name and the unique ID of the object attached to.
signal_objectsDetached	Called when the observer detaches from all. Includes the observer name.
signal_linearScaledSpeedGain Changed	Called when the scaled linear speed gain is changed. Includes a pointer to the observer and the new gain.
signal_linearUnscaledSpeedGain Changed	Called when the unscaled linear speed gain is changed. Includes a pointer to the observer and the new gain.

Table 4-2: Observer signals

Signal	Description
signal_rotationSpeedGainChanged	Called when the rotation speed gain is changed. Includes a pointer to the observer and the new gain.
signal_orbitSpeedGainChanged	Called when the orbit speed gain is changed. Includes a pointer to the observer and the new gain.
signal_gainPrecisionChanged	Called when the gain precision is changed. Includes a pointer to the observer and the new gain.
signal_terrainConstraintChanged	Called when terrain constraint is turned on or off. Includes a pointer to the observer and a boolean indicating whether terrain constraint is on or off.
signal_speedScalingChanged	Called when speed scaling is turned on or off. Includes a pointer to the observer and a boolean indicating whether speed scaling is on or off.

4.5. The Observer API

Application code can access an observer by:

- Connecting to the signal_observerCreated from the *DtInputDriverSignaler*
- Using one of the *DtInputDriver's* accessors to retrieve the current observer, or any observer by its name.

Both of these mechanisms result in a pointer to a *DtObserver*.

The *DtObserver* class is a facade for the *DtObserverObjectAgent*, which is itself the interface for the distributed *DtObserverObjects* it controls. Calls to *DtObserverObjectAgent* methods are distributed to all of the agent's *DtObserverObjects*. The *DtObserver* class is intended to simplify some operations on observers. However, it is always possible to access the agent itself from the *DtObserver* and make calls directly. There are many methods that you can call to control an observer.

4.5.1. Controlling the Observer

DtObserverObjects are controlled in the local (database) coordinate system. Given a pointer to an observer, pObserver, a call to set a *DtObserverObject's* location would look like:

```
pObserver->setLocation( desiredX, desiredY, desiredZ );
```

You can specify orientation using Tait-Bryan angles or quaternions:

```
pObserver->setOrientation( desiredPsi, desiredTheta, desiredPhi );
pObserver->setOrientation( xQuat, yQuat, zQuat, wQuat );
```

Since many VR-Vantage customers may need to work in geocentric coordinates, *DtObserver* also provides methods that convert from geocentric to local coordinates before distributing the commands to the *DtObserverObject*:

```
pObserver->setGeocentricLocation( geocX, geocY, geocZ );
pObserver->setGeocentricOrientation( geocPsi, geocTheta, geocPhi );
```

Methods also exist for setting attachments, attach modes, and all the parameters typically associated with those modes.

The keyboard-based navigation code does not set the location and orientation of the observer directly. It uses commands to start and stop observer movement. These methods are not part of the *DtObserver* facade. You can call them directly on the *DtObserverObjectAgent*. For example:

```
pObserver->agent()->startMoveForward();
pObserver->agent()->stopMoveForward();
```

You can retrieve information about the observer's location, orientation, current attachments and modes, and so on. Please see the class documentation for *DtObserver* and *DtObserverObject* for all the methods available.

Drivers

This chapter explains the function of drivers in VR-Vantage and the major drivers supplied.

5.1. Drivers

As explained in "Architectural Overview," on page 2-7, the Driver layer contains the application logic. Drivers take input from users and the simulation network and send instructions to the VR-Vantage Control Interface. To display data in a scene, you must write a driver that creates visual objects using domain-specific logic, user input, or simulation data.

Composing a scene is the job of a driver. A driver connects logic, data, and events with the visual system. Logic could be a local high fidelity physics simulation, events could include keyboard and mouse input, and data could be the internal terrain representation. All three could be from a networked, distributed simulation.

Each driver is designed to be modular and may be user-controllable. Drivers can be added and removed from the display engine once it is initialized. The display engine can support many drivers.

5.1.1. Starting and Stopping Drivers

A driver can be in a stopped state or a running state. Initially a driver is in the stopped state. A stopped driver is functionally inactive.

Drivers can be started and stopped any number of times. When a driver is started it creates the visual resources it will update. While a driver is running, it creates, destroys, and updates these visual resources. When a driver stops, it must deallocate all allocated visual resources. It must restore the scene to the state that existed before the driver was started. This capability is used to reload the scene when connecting to display engines. (For details, please see "Distributed Objects and Agents," on page 4-7.)

You can set drivers to automatically start and stop based upon the existence of windows. When a driver is running it is constantly ticked at the rate of the display engine. To ensure good performance, a driver should try to do as little work as possible in its tick() function.

The display engine supports many simultaneous running drivers. The use, configuration, and runtime behavior of drivers is application-specific. Some drivers are built into the display engine, while others are available from other VR-Vantage modules. Two important built-in drivers are the scene driver and the input driver.

5.2. The Scene Driver (DtSceneDriver)

The scene driver, ***DtSceneDriver***, is the default logic for the built-in scene logic for the display engine. The scene driver owns a scene (***DtScene***. It can load and save the scene using the VR-Vantage scene file format. The scene driver is automatically started when the display engine is initialized. It does not perform any actions during tick(). (For a description of the capabilities of the scene the driver hosts, please see Chapter 6, *Composing Scenes*.) You can extend the scene driver class by replacing the corresponding creator instance. The VR-Vantage Toolkit assumes there is a valid scene driver at all times.

Typically, VR-Vantage applications get their data from networked simulations or hosts. VR-Vantage is designed for this and most of the time the correct place for an agent to get created is in a driver. VR-Vantage includes drivers for HLA 1.3, HLA 1516, CIGI, and DIS. For more information, please see "Simulation Drivers," on page 5-8.

5.3. The Input Driver

The input driver, (***DtInputDriver***, owns the:

- Observer objects.
- Mouse and keyboard event listener (***DtKeyAndMouseEventListener***).
- Key Map Manager (***DtKeyMapManager***).
- Observer controller (***DtObserverController***).

Like the scene driver, the input driver is started automatically when the display engine is initialized.

The input driver creates observers as requested, processes mouse events, processes key events using the Key Map Manager, and controls the observer based on the mouse and key events. It also interacts with the Selection Manager (***DtSelectionManager*** when processing certain mouse events.

When the input driver is created, it creates the Key Map Manager. It creates the observer controller and the key and mouse event listener when it starts and destroys them when it stops.

5.3.1. Creating Observer Configurations

The input driver maintains a list of observer configurations (*DtObserverConfiguration*) that describe the observers that should be present when the input driver is running. A list of observer configurations is provided to the input driver when it is created. Additional configurations can be added. If a configuration is added while the input driver is running, then that observer is created immediately. Otherwise, it is created the next time (and each subsequent time) the input driver is started. The following example shows how to add an observer configuration to the input driver.

1. Create an observer configuration to describe the observer to be created. The observer configuration lets you change many of the default values for the observer, but all you really need to specify is the new observer's name in the constructor:

   ```
   makVrv::DtObserverConfiguration observerConfig("Example Observer");
   ```

2. Add the configuration to the input driver's observer configuration list:

   ```
   de.driverManager().inputDriver().addObserverConfiguration(
     observerConfig );
   ```

By adding the observer configuration to the input driver's observer configuration list, you ensure that the observer will be created each time the input driver is started.

The **exampleCreateObserver** example shows how to create a complete VR-Vantage application with an additional observer created as described in this section. With a second observer, you can open the display configuration editor, select a channel, and change the observer associated with that channel.

5.3.2. Input Driver Signals

The input driver uses the *DtInputDriverSignaler* to notify other objects of the operations that it performs. Table 5-1 describes the input driver signals.

Table 5-1: *DtInputDriverSignaler* signals

Signal	Description
signal_inputDriverCreated	Emitted when an input driver is created. Provides a pointer to the input driver selected.
signal_inputDriverAboutToBe-Destroyed	Emitted when an input driver is about to be destroyed. Provides a pointer to the input driver about to be destroyed.
signal_observerCreated	Emitted when an observer is created. Provides a pointer to the input driver and a pointer to the observer created.
signal_observerAboutToBe-Destroyed	Emitted when an observer is about to be destroyed. Provides a pointer to the input driver and a pointer to the observer about to be destroyed.

Table 5-1: *DtInputDriverSignaler* signals

Signal	Description
signal_currentObserverChanged	Emitted when the current observer changes. Provides a pointer to the input driver and a pointer to the new observer.
signal_rightClickTerrainMenu-Request	Emitted when the user right-clicks on the terrain. Provides the <x,y> mouse coordinates and a pointer to the channel that was moused in.
signal_rightClickSelectionMenu Request	Emitted when the user right-clicks on an object in the scene. Provides the <x,y> mouse coordinates and a pointer to the channel in which the mouse was clicked.
signal_entityIndicatorsToggled	Emitted when entity indicators are toggled.
signal_viewControlsEnable	Emitted when the Accept View Controls switch is changed. Provides a boolean to indicate whether view control is on or off and a pointer to the affected observer.

5.3.3. The Keyboard and Mouse Listener

Management of keyboard and mouse events for an application is often dependent on which GUI system (if any) is in use. Sometimes other components within an application take responsibility for handling these events. For example, OSG provides a keyboard and mouse event handling system.

The keyboard and mouse event listener provides an interface for handling mouse and keyboard events that is independent of the GUI, the renderer, or other components that may grab events. The base class, ***DtKeyAndMouseEventListener***, is a mechanism for registering and creating the type of listener needed for the system in use. It maintains accessors for the *DtDe* and the input driver, which provides a pointer to itself when it creates the listener. The listener is expected to process keyboard and mouse events in whatever manner it chooses, convert them to the VR-Vantage-specific ***DtKeyEvent*** and ***DtMouseEvent*** classes, and pass these on to the input driver through its processKeyEvent() and processMouseEvent() methods.

Since VR-Vantage uses an OSG-based renderer, it uses the ***DtOsgKeyAndMouseEventListener***, which gets the keyboard and mouse events from an OSG event queue, converts them to *DtKeyEvents* and *DtMouseEvents*, and passes them on to the input driver. Each OSG channel has its own event queue. If there are multiple observers, events will affect whichever observer is assigned to the channel in which the events occur.

When a Qt-based GUI is in use (as it is in MÄK's VR-Vantage applications), Qt grabs the keyboard and mouse events before they get to OSG. VR-Vantage uses the *DtOsgAdapterWidget* to convert these events to OSG keyboard and mouse events, and then puts them on the correct OSG event queue for the channel in which they occurred.

5.3.4. The Observer Controller

The observer controller tracks which observer is the one currently under keyboard and mouse control, and works with the Key Map Manager to execute observer functions on specific keyboard events. It provides mappable functions that wrap the observer functions that we want to be able to invoke from the keyboard. Mappable functions take no arguments and return void. The observer controller registers each of these functions and a corresponding string name with the Key Map Manager. Once registered, key events can be bound to the string names as part of a key map, which in turn causes the registered functions to be executed when those key events occur.

5.3.5. The Key Map Manager

The Key Map Manager maintains a list of all functions that can be bound to a key and the string representation for that function. It also maintains a list of key maps, which provide mappings from key events to those string representations. The string representations are used to find and invoke the desired function. Please see the **exampleKeyMap** example in the class documentation for an explanation of how to register a function with the Key Map Manager, and bind it to a key event in a key map.

The Key Map Manager uses the Key Map Manager signaler (***DtKeyMapManagerSignaler*** to emit signals when certain operations are performed. Table 5-2 describes the *DtKeyMapManagerSignaler* signals.

Table 5-2: *DtKeyMapManagerSignaler* signals

Signal	Description
signal_keyMapCreated	Emitted when a key map is created. Provides a pointer to the Key Map Manager and the name of the key map created.
signal_keyMapAboutToBeDestroyed	Emitted when a key map is about to be destroyed. Provides a pointer to the Key Map Manager and the name of the key map about to be destroyed.
signal_keyMapChanged	Emitted when the current key map in use changes. Provides a pointer to the Key Map Manager and the name of the newly current key map.

5.3.6. Custom Event Processors

The input driver provides a way to see (and optionally, handle) keyboard and mouse events before it processes them. It maintains a queue of event processors, which are derived from the abstract base class ***DtEventProcessorInterface***. You can add event processors to the front or the back of the queue. When a key or mouse event is received, the input driver invokes the appropriate processKeyboardEvent() or processMouseEvent() method for the event processor on the front of the queue. If the event processor wants to handle the event, it can return true; otherwise it returns false and the next processor on the queue is invoked. When the queue of event processors has been exhausted, if none of them has handled the event, the input driver handles the event directly.

The **exampleEventProcessor** example shows how to create an event processor. For details, please see the class documentation.

5.4. Simulation Drivers

The VR-Vantage Toolkit provides drivers for several protocols in the protocol-specific modules. Each driver provides the same set of capabilities, with minor exceptions based upon the protocol. These drivers visualize entities and interaction effects such as detonations. They also handle advanced visualization features such as track histories. You can have multiple simulation drivers added to the display engine, each with a different configuration. Each instance of a simulation driver is referred to as a simulation connection. You can save exercise configurations and easily switch between connections.

5.4.1. Stopping and Starting Simulation Drivers

A simulation driver's state is controlled by its connection state. When a simulation driver is not connected to a simulation, it is in the stopped state. A simulation driver enters the running state (starts) when it successfully connects to a distributed exercise. While the simulation driver is connected to the exercise it continues to run. Disconnecting from the exercise causes the driver to stop.

5.4.2. Using Shared Source Files for Simulation Drivers

VR-Vantage simulation drivers use VR-Link, which provides a single interface for multiple simulation protocols. Because the interface is largely the same for both DIS and HLA, almost all protocol-specific classes are written once and compiled multiple times (for each simulation protocol). Each class is compiled into a separate namespace to eliminate duplicate symbols. Shared source classes are written in header (*.h*) files and inline (*.inl*) files. However, the inline files are not included by the header files. All shared source inline files are included in a single source (*.cxx*) file to be compiled for the particular library that defines the namespace.

5.4.3. Simulation Driver Classes and Namespaces

The functionality of the distributed simulation drivers is implemented in a small number of associated classes and patterns in addition to the primary driver class.

To distinguish between classes with the same name, a different namespace is used under the main makVrv namespace. For example the ***DtVrlinkConnection*** class represents the simulation connection. The DIS protocol library is call vrvDis. The complete name of this class is *makVrv::vrvDis::DtVrlinkConnection*. In the header and inline files the namespace is defined using a macro, DT_PROTOCOL_NAMESPACE, which is redefined for each library that recompiles the code.

5.4.4. Connections

Each driver class has a connection class called ***DtVrlinkConnection***, which manages the VR-Link exercise connection and registers listener classes. The connection class manages ***DtVirtualBaseClass*** instances. It deletes all instances before disconnecting by deleting the exercise connection at the appropriate time.

A connection registers listeners.

5.4.5. Listeners

A listener class provides signals for a particular type of data. The templated class ***DtInteractionListener*** can listen to any VR-Link interaction class. The *DtFireInteractionListener* is an example of a specialization that listens for instances of the ***DtFireInteraction*** class to be received. Stateful objects such as entities have two levels of listeners. The main instance listener class ***DtEntityExistenceListener*** listens for newly discovered entity instances and signals when existing instances are deleted. For each entity that is discovered, there is a ***DtEntityStateListener***, which listens for changes to a particular entity's state and signals when attributes change. Listeners are created lazily on demand from a visualizer or a processor. Listener classes inherit the ***DtVirtualBaseClass*** and should be registered with the connection to ensure that they are destroyed before the exercise connection is destroyed.

Typically there is only one instance of a listener, because instances are registered with the connection instance of a driver. Individual state listeners are an exception. They are owned by the instance listener, which follows this rule.

5.4.6. Visualizers and Processors

Visualizer classes and processor classes use listeners to make visualization calls or data processing calls. Visualizer classes do not follow any pattern or class hierarchy other than that they connect to the signals of a particular listener and create scene object and model agents, which interact with the scene. An example of a visualizer is the ***DtFireInteractionVisualizer***, which creates a muzzle flash effect in the scene for each *DtFireInteraction* that is received. A processor class is similar to a visualizer, but it does not make agents. An example of a processor is the *DtViewControlProcessor*, which receives custom Stealth Control messages and makes calls that control VR-Vantage's observers.

Usually there is only one instance of a particular type of visualizer or processor. While not required, it is useful to register the instance of the visualizer or processor with the connection to ensure proper memory management. Visualizers and processors can be dynamically added and removed from a connection. The Driver Accessories pattern lets you add visualizers to existing driver instances, thereby extending the functionality without modifying the existing capabilities.

5.5. Adding Functionality to a Driver (Accessories)

Driver accessories allow functionality to be added to a driver without completely replacing it. Each driver can be written to support different amounts and types of extensibility through creators or signals. The VR-Vantage Toolkit provides a common mechanism for installing additional functionality on a per-driver basis through the use of driver accessories. A driver accessory is a simple interface for executing code when a driver starts. An accessory is registered with the display engine and can support any driver class that it knows how to interface with. The simulation drivers fully support modular extensibility using accessories.

Composing Scenes

Scenes are made up of terrain, models, and the environment.

6.1. Scene Composition

A scene contains an environment, the time of day, and possibly, a terrain. Scenes are composed on the master display engine by creating a set of distributed visualization objects, which are then created (typically as OSG nodes) on each distributed display engine node. The basic visualization object is the model (*DtModel*, each of which is owned by a *DtSceneObject*. One scene object can contain any number of models and provides an interface for positioning and orienting them as a single unit. The distributed scene objects are in turn owned by a variety of facade objects that organize them and provide convenient interfaces to drive and manipulate their appearance. This chapter summarizes the various types of models, scene objects, and their containers, and how to use them to construct scenes.

Data is organized into a structure called a scene graph. To add objects to the scene, you add nodes to the scene graph. For an example of how to add a node to the scene graph, please see the **exampleSceneRoot** example.

Although you can modify the scene graph as shown in exampleSceneRoot, this is not actually the preferred method. The preferred method is to use the control layer. It modifies the renderer of both the local and any remote display engines. The is called distributed rendering, which synchronizes the scene on all display engines. For details about distributed rendering, please see "Distributed Rendering," on page 1-6.

6.2. Models, Model Instances, and Model Definitions

A *DtModel* is the basic unit of scene composition. One model corresponds to one renderable object on each display engine. For example, visualizing a tank might involve one model for the tank's geometry, one model for its dust cloud, and another to display its track history. The basic types of models in VR-Vantage, and their uses are:

- *DtArticulatedModel* – Displays 3D models with articulated parts
- *DtCharacterModel* – Displays human characters, typically DI-Guy characters
- *DtDrawModel* – Draws lines in the scene programmatically
- *DtParticleSystemModel* – Creates various special effects, such as treadmarks and wakes.
- *DtAnimationModel* – Creates animations, such as detonation effects.
- *DtModel* – A general purpose model interface
- *DtTreeModel* – Displays trees in a *DtForest*, typically SpeedTrees.

The details required to draw a model, such as geometry, filename, scale, or texture are provided by a ***DtModelDefinition***. A model definition is a set of parameters, whose names and types are defined by a ***DtSchema***. The ***DtModelDefinitionManager*** saves and loads model definitions and schemas and maintains settings for model instancing and caching. Model definitions and schemas are distributed, but not by the usual agent mechanism. Instead, the ***DtDeSharedState*** distributes them directly.

A *DtModel* is only the distributed interface of a renderable scene element. It does not directly modify the scene graph of a display engine. Instead, when the model is given a model definition through its setModelDefinition() method, it creates a local ***DtModelInstance*** on each display engine, which is then responsible for putting the appropriate nodes in that display engine's scene graph. The attributes of the model instance are determined by the model definition's parameters and schema type.

Model instance types follow the Creator pattern. (For information about the Creator pattern, please see "The Creator Pattern," on page 9-10.) Each model instance type has a creator registered with the ***DtModelInstanceFactory***, which the model uses to generate its instance. Once created, the instance sets itself up according to the model definition. The setup can either be from scratch, which typically means importing OSG nodes from a file, or if model instancing is enabled, by copying a cached instance from the model definition's local instance cache. When the model is added to a scene object, the instance inserts the appropriate nodes into the scene graph, usually (but not always) under the scene object's position and orientation transform nodes.

6.3. Terrain Structure

A terrain (***DtTerrain***) in VR-Vantage is a facade around a set of ***DtTerrainPatch*** and ***DtFeatureLayer*** objects. The terrain maintains records of each operation performed on it (such as adding or removing a terrain patch), which are saved and loaded to MTF files. The *DtTerrain* is not a distributed object, so it exists only on the master display engine. If you are using external display engines, each display engine must have the same terrain files on its local computer.

6.3.1. Terrain Patches

A *DtTerrainPatch* is a facade around a ***DtTerrainPatchObject*** , which is a distributed object that combines the functionality of a model and a scene object. The terrain patch maintains a history of operations performed on it, such as extracting props or adding an overlay image. This history is included when the terrain is saved and executed when it is loaded. Each *DtTerrainPatchObject* has a model definition that specifies where to find the geometry (typically through the **filename** parameter) and a coordinate system (through the **terrainCoordinateSystem** and **terrainCoordinateSystemParameters** parameters).

The coordinate system of the most recently loaded terrain patch is always used as the coordinate system of the entire scene. In contrast to other scene objects, a *DtTerrainPatchObject* has no position or orientation, since all other objects in the scene are transformed in its coordinate system. This may cause undesired behavior if multiple terrain patches with different coordinate systems are loaded into the same scene.

Terrain patches are located in the scene graph under the terrain root node. They are included in intersection tests for ground clamping and navigation, but excluded for selection.

6.3.2. Feature Layers

A *DtFeatureLayer* is a facade around a collection of ***DtProp*** objects, which represent distinct terrain-like features such as lampposts and buildings. Feature layers are created in VR-Vantage by:

- Loading point features and mapping them to models
- Extracting geometry out of a terrain and converting it into a model.

6.3.3. Loading Point Features

VR-Vantage provides the convenience class ***DtPropAdder*** to load point features from files. The prop adder finds point features that match a rule specified by a ***DtAddPropsRuleRecord***. Then it creates a prop at the location of each point feature, using the model definition specified in the rule. In order for the feature layer to correctly maintain its state, the *DtPropAdder* should not be used directly. Use the *DtFeatureLayer*'s addProps() method, which makes the correct calls on the *DtPropAdder* for you. For instance, suppose you wanted to add SpeedTrees at the sites of tree point features in a GDB file. You could:

1. Add the GDB to the *DtTerrain* as a feature layer (*DtTerrain*::addFeatureLayer()).
2. Create a DtAddPropsRuleRecord that maps a tree type to an appropriate model definition (for example, EC030 to TreesEuropeanWhiteBirchST).
3. Pass it to the feature layer (DtFeatureLayer::addProps()).

6.3.4. Extracting Geometry

A *DtTerrainPatch* can find nodes in its scene graph and move them into *DtModel* objects. The *DtModel* for each prop must be created in the driver to be distributed to all display engines, but creating them requires knowledge of the scene graph. Therefore, basic prop extraction happens in a multi-step process:

1. The *DtTerrainPatch* makes a call to extract nodes out of the scene graph and into a list on each display engine. (The nodes must match a search expression.)
2. The master display engine signals the *DtTerrainPatch* with the number of extracted nodes.
3. The *DtTerrainPatch* creates that number of props (including *DtModels*), and passes each model's object ID along with an index back to the display engines.
4. The display engines create local model instances for each model and move the extracted node with the corresponding index into each one.
5. The node search expressions check the name, filename, and description fields of each node for a match. They can use either regular expressions or plain search strings.

6.4. Environmental Effects and the Lighting Model

VR-Vantage implements environmental and lighting effects through SilverLining. It provides two interfaces to set SilverLining state, the ***DtDeTimeManager*** and the ***DtEnvironment***. The *DtDeTimeManager* controls date and time. The *DtEnvironment* controls SilverLining's clouds, precipitation, air turbidity, and visibility effects.

6.4.1. Managing Clouds, Visibility, and Precipitation

The *DtEnvironment* is a member of a *DtScene*. It provides a serializable interface to SilverLining environmental effects and it maintains records of its state, which can be saved or loaded with the makArchives::DtEnvironmentRecord. A *DtEnvironment* wraps a ***DtSkyObject***, which is the distributed interface to the SilverLining clouds, precipitation, turbidity, and visibility effects. To set your scene's environmental effects, you might do something like the following:

```
scene.environment().clear();
scene.environment().setPrecipitationEnabled(true);
scene.environment().setPrecipitationType(DtEnvironment::Rain);
scene.environment().setPrecipitationIntensity(0.5);
DtCloudLayerRecord cloudLayer;
cloudLayer.setType("CumulusCongestus");
cloudLayer.setEnabledFlag(true);
cloudLayer.setAltitude(1000);
cloudLayer.setWidth(20000);
cloudLayer.setLength(20000);
cloudLayer.setThickness(100);
cloudLayer.setDensity(1.0);
scene.environment().addCloudLayer(cloudLayer);
```

6.4.2. Managing the Time and Date

The *DtDeTimeManager* maintains the scene's time and date, as well as changing the time each tick according to its time scale setting. The *DtDeTimeManager's* settings are distributed to all nodes by the *DtDeSharedState*. This means that you should never modify the *DtDeTimeManager* directly; instead, use the functions DtDeShared-State::setDateAndTime() and DtDeSharedState::setTimeScale(). The date and time are the main determinants of the VR-Vantage lighting model's state.

6.5. Forests

SpeedTrees in VR-Vantage are implemented as members of a single SpeedTree forest on each display engine. The *DtForest* is a facade around the distributed *DtForestObject*. The *DtForestObject*, like the *DtTerrainPatchObject*, combines some of the functionality of a scene object and a model into a single object.

The forest is located in the scene graph under the object root node, but since Speed-Trees are drawn outside of OSG, they are not included in any intersection tests.

6.6. Scene Objects

A scene object (*DtSceneObject*) groups together models with shared spatial data and represents one conceptual element in the scene within one model set. For example, one scene object could represent a single tree in the 3D Models model set, or the main model, flames, and trajectory history line associated with a tank. Scene objects track the element ID of the simulated element they represent.

Each scene object contains a vector of positions and orientations associated with it. It is up to the scene object's models to determine how those kinematic parameters are used. For example, a simple model might use its scene object's first position and orientation values to position and orient itself in the scene, while a polygon model might use several of the scene object's positions to define its vertices.

A scene object is given a model set when it is created. When adding a model to a scene object, the model's visual type is also specified. Available visual types are defined in the visual type manager (*DtVisualTypeManager*). The combination of model set and visual type is used to determine which models to render on a particular channel. For example, a model added to a scene object in the XR model set with the visual type "dust_clouds" is rendered only on channels that are displaying the XR model set and have dust clouds turned on. VR-Vantage implements this render control by creating a root node in the scene graph for each visual type in a model set (Figure 6-1). The nodes for each model are placed under the appropriate root node, and during a channel's render traversal the roots that should not be rendered are skipped.

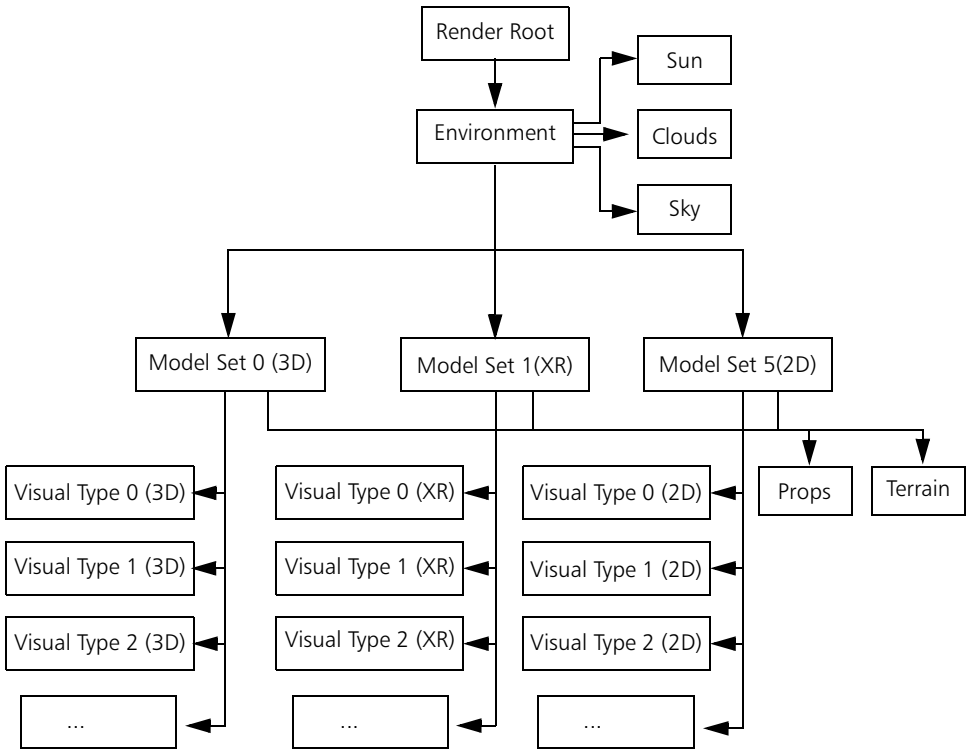


Figure 6-1. Scene graph

If a scene object's spatial data changes frequently, it may have an updater (*DtUpdater*) associated with it. The updater is responsible for any changes that need to be made to the scene object (usually position and orientation) on a per-frame basis, such as dead reckoning and ground clamping.

6.7. Entities

Entities are represented in the scene by scene objects, to which models are attached. The VR-Vantage Toolkit provides some common kinds of entity primitives. These are called facades and show how models and model instances are added to a scene object.

6.7.1. Placing Entities in a Scene Using a *DtVrlinkConnection*

Although you can place individual entities in a scene using the ***DtEntityFacade***, it is common to use a ***DtVrlinkConnection*** (which is created by one of the drivers based on the *DtVrlinkDriver*) to populate a scene. This section describes how entities are created within a connection.

Individual entities are handled by the ***DtEntityElementProcessor***. The *DtEntityElementProcessor* is registered as a singleton instance with the *DtVrlinkConnection*. The *DtEntityElementProcessor* uses (or creates if it does not exist) a singleton instance of a ***DtEntityExistenceListener*** registered with the connection as "DtEntityListener" at construction; a different (derived) existence listener may be registered, but this must occur before the creation of the *DtEntityElementProcessor*. The *DtEntityElementProcessor* also looks up the ***DtVisualizerCreationSettings*** instance registered with the connection, which will be used to determine which visualizers need to be created when an entity is discovered (as described in "The Visualizer Creation Settings Manager," on page 6-10). It also connects to the signal_visualizerCreationEnabled and signal_visualizerCreationDisabled signals of the ***DtVisualizerCreationSettingsSignaler***, which tells the *DtEntityElementProcessor* when visualizers need to be created or deleted for entities, as well as allowing it to keep the *DtVisualizerCreationSettings* up-to-date.

Creating Entities

When the *DtEntityExistenceListener* discovers a new entity, it allocates a ***DtVrlinkEntityStateListener*** for use with the entity, which in turn allocates a unique ID for that entity. The *DtEntityExistenceListener* sends a signal to the *DtEntityElementProcessor* informing it about the new entity, and includes the new *DtVrlinkEntityStateListener* as part of that signal. The *DtEntityElementProcessor* then creates and initializes a ***DtEntityElement***, using the unique ID allocated by the *DtVrlinkEntityStateListener* as the entity's DtElementID. The *DtElementID* can be used to look up the *DtVrlinkEntityStateListener* or the *DtEntityElement* for that entity. The new *DtElementID* is registered with the *DtEntityElementProcessor's* ***DtAggregateEchelonManager***, and finally, the *DtEntityElementProcessor* raises its signal_elementCreated signal (which includes both the new *DtEntityElement* and its associated *DtVrlinkEntityStateListener*).

When constructed, the *DtEntityElement* looks up the name of the element definition for its entity type. When it is initialized by the *DtEntityElementProcessor* (by calling DtEntityElement::visualize()), it looks up the *DtEntityElementProcessor* instance (in the connection) and, for each model set that can be visualized (please see "Scene Objects," on page 6-6 for a discussion of model sets), creates a ***DtEntityVisualizerSet***. Each *DtEntityVisualizerSet* is in turn initialized by calling DtEntityVisualizerSet::visualize() with the name of the element definition to use and the *DtVisualizerCreationSettings* from the *DtEntityElementProcessor*.

DtEntityVisualizerSet::visualize() looks up the ***DtElementDefinitionTypeManager*** instance (also in the connection), and then looks up the ***DtElementDefinitionNew*** for the element definition argument in the *DtElementDefinitionTypeManager*. It then calls its base class DtVisualizerSet::visualize() method with the element definition name, the *DtElementDefinitionNew* that was looked up, and the *DtVisualizerCreationSettings* from the *DtEntityElementProcessor*.

DtElementDefinitionNew contains a ***DtVisualizerDefinitionSet*** for every model set defined. Since the *DtVisualizerSet* was initialized with its model set specified, it can find the ***DtVisualDefinitionSet*** for the model set it needs to visualize. The *DtVisualizerDefinitionSet* is a list of ***DtVisualizerDefinitions***, that describe the visualizers that exist for the entity in this model set. The *DtVisualizerSet* iterates through the *DtVisualizerDefinitionSet*, and if the *DtVisualizerCreationSettings* indicates that visualizers with the visual type of a *DtVisualizerDefinition*, that *DtVisualDefinition* will be realized into some type of ***DtStateVisualizer***, using the ***DtStateVisualizerFactory***.

DtVisualizerCreationSettings indicate that a visualizer should not be created for a given model set only if that visualizer has a visual type that has been specified in the DtVisualizerCreationSettingsManager::VisualTypeVisualizerCreationStrategyMap(see) as being 'create-as-needed' and that visual type is not currently being displayed for any observer using that model set. Note that many visualizers have no real defined visual type (and so have their visual type set to 0). These visualizers will always be created. The defined visual types are:

- main_model
- dust_clouds
- wakes
- contrails
- footprints
- treads
- fire
- smoke_plumes
- detonations
- muzzle_flashes
- track_histories.

By default, these all have the 'create-as-needed' strategy. The visual type is assigned to a visualizer definition (and on the *DtStateVisualizer* that realizes it) in the Visual Definition Editor.

The Visualizer Creation Settings Manager

The *DtVisualizerCreationSettingsManager* maintains a map of visualizer creation strategy (either create-always or create-as needed) by visual type. It can also provide, on request, a map of which visualizers, based on visual type and model set, should be created given that visual type's creation strategy and the current observer settings in use. That map is wrapped in a *DtVirtualBaseClass* called *DtVisualizerCreationSettings*. So if, for example, the creation strategy for visual type "track_histories" is create-as-needed and none of the observers that are currently using an observer mode that both uses model set 0 and has track histories enabled, then visualizers with the "track_histories" visual type do not need to be created for model set 0.

When a new connection is created, it will ask the *DtVisualizerCreationSettingsManager* to create and initialize a *DtVisualizerCreationSettings* object and register it with the connection. It will then be available for the *DtEntityElementProcessor* that the connection will subsequently create, providing it with an up-to-date initial state. The *DtVisualizerCreationSettingsManager* also listens to changes to observer mode selections and observer mode model set and settings changes, and generates signals whenever it becomes appropriate to enable or disable the creation of a specific visual type for a given model set. The *DtEntityElementProcessor* listens for these signals and updates its *DtVisualizerCreationSettings*, as well as creating or destroying visualizers as needed for entities that already exist.

6.7.2. Aggregate Entities

To visualize aggregates in a VR-Vantage application, you must first be connected to a communications protocol. This can be DIS, HLA13, or HLA1516 and occurs in the respective simulation driver. Once connected, there must be traffic on that network that VR-Vantage can identify as an aggregate. This happens when an application publishes an Aggregate State PDU on the network. When VR-Vantage recognizes this PDU, it internally maps the entity type of the aggregate to an aggregate element definition.

A visual definition is a series of visual components that are used to visualize an object on the 2D overlay or as 3D geometry. An aggregate visual definition will usually contain the following visualizers:

- A main 2525B icon (the Military Symbol Icon Visualizer)
- A way to position the icon in the database (Kinematic State Visualizer)
- The bounding extents of the aggregate (Aggregate Extent Visualizer)
- The selection rectangle (Selection Visualizer).

The *DtVrlinkAggregateExistenceListener* listens to the network. For each aggregate, it communicates with the ***DtAggregateElementProcessor*** to create new aggregate visuals. The *DtAggregateElementProcessor*, when notified of a new aggregate, creates a ***DtAggregateElement***. For each visualized model set, the *DtAggregateElement* creates a ***DtAggregateVisualizerSet***. This *DtAggregateVisualizerSet* uses the supplied visual definition set (as discovered from the aggregate type mapping) to visualize the elements contained within.

The *DtAggregateElementProcessor* also uses the ***DtAggregateEchelonManager*** class to manage the echelon hierarchy. The *DtAggregateEchelonManager* manipulates a ***DtElementIDHierarchy*** in order to keep parent/child relationships of elements that are created in the system. The *DtAggregateElementProcessor* is also responsible for hiding and showing aggregate elements based on the ***DtAggregateSettingsRecord*** as well as the expanded/collapsed state of the aggregate. This expanded/collapsed state is communicated from the UI thread to the simulation driver thread via element attributes, specifically the DtElementIDHierarchyDisplayStateAttribute attribute.

6.7.3. Sensor Volumes

As the entities are placed in the scene, the ones that support emitter systems are created with sensor volume visualizers (*DtVrlinkEntitySensorVolumeVisualizer* for 3D model sets, and *DtVrlink2DEntitySensorVolumeVisualizer* for 2D model sets). These visualizers listen for the creation and destruction of emitter systems by connecting to the signals emitted by the emitter system existence listener (*DtEmitterSystemExistenceListener*).

When an emitter system (*DtEmitterSystemRepository*) is created, a custom implementation is used (*DtVrlinkEmitterStateInterceptor*) to intercept calls that change the state of the emitter system and signals the emitter system's state listener (*DtVrlinkEmitterStateListener*). Visualizers now connect to the additional signals provided by the state listener (for more detailed information about the syste's emitter beams and overall configuration), and creates a visual representation for all emitter beams already attached to the discovered emitter system. As new emitter beams are discovered, or when existing emitter beams change, the visualizer updates the sensor volume visualization accordingly.

6.8. Props

VR-Vantage props are scene elements that act like pieces of terrain, but can be individually manipulated. Props are owned by a *DtTerrainLayer*, which is an interface implemented by the *DtFeatureLayer* and *DtTerrainPatch*. The *DtProp* class is a facade around a *DtSceneObject* and a *DtModel*. It maintains state information on the master side for saving and loading and GUI display.

6.9. Effects

The *DtEffectModel* is used to display a variety of special effects in VR-Vantage, such as flames and track histories. Effects come in three major types: ribbon and decal effects, particle effects, and animation effects. The type of effect is determined by the schema used for its model definition. The *DtEffectModelInstance* creates different types of geometry depending on the schema name and parameters provided.

6.9.1. Ribbon Effects

A ribbon effect is a flat "ribbon" that stretches out behind a moving scene object. It can be colored, textured, faded out, and moved, depending on the model definition schema used to define it. The OSG node classes in vrvOsg that inherit ***DtOsgRibbon*** are all implementations of ribbon effect variants. Ribbons are located in the scene graph under the effect root node and contain a reference to the scene object that owns them. At each frame, the ribbon finds the location of its scene object and extends itself, if necessary.

The most basic ribbon effect is the ***DtOsgTrackHistoryRibbon***, which is a simple colored quad strip with a line along each edge. It is created for the ribbon model schema. Track history effects position and size themselves automatically according to the bounding box of the scene object that they are attached to. Therefore, it is important to add track history effects to a scene object after its main model is added. Track history ribbons have a maximum length parameter to prevent them from growing too large for fast-moving scene objects. Quads that are beyond this length are quickly faded out.

The ***DtOsgDecalRibbon*** is similar to the track history ribbon, with the addition of a texture. The width and relative position of a decal ribbon must be specified in its model definition, with the PointA/PointB parameters. If PointC/PointD are also specified, a ***DtOsgMultiRibbon*** is created instead, which is a composite of two decal ribbons. This technique is used, for example, to create the default missile trail effect. Like track history ribbons, decal ribbons use a maximum length to limit their size.

The most complex ribbon effect is the ***DtOsgParticleRibbon***. Particle ribbons look similar to decal ribbons, with the addition of motion and lifetime-based fadeout. The **ParticleVelocity** parameter specifies the speed in meters per second at which each horizontal pair of points moves apart. The **AlphaVelocity** parameter sets the change in segment alpha per second, and thereby determines the time in seconds that it takes for a pair of points to fade out. The particle ribbon is used for the default wake effect, which spreads and fades out over time.

6.9.2. Particle Effects

VR-Vantage uses the osgParticle package to create particle effects. Each particle effect model instance has a particle emitter node that is added to the parent scene object, and a particle geometry node that is placed under the effects root. This allows the particles, once emitted, to move independently from their parent scene object. Particle effects are usually loaded from files. These files can be created programmatically or with an editor such as the Delta3D particle system editor. Particle effects use the effect schema.

6.9.3. Animated Effects

Animated effects are used by VR-Vantage to create particle-like effects with less use of resources. An animated effect is created by setting the **filename** parameter in an effect with the effect schema. This file is loaded at the location of the effect's parent scene object. Animated billboards are loaded as animated effects for VR-Vantage's default detonations.

6.10. Grid Lines

VR-Vantage draws grid lines on a per-channel basis. Channel grid lines use the *DtChannelKeywordObject* mechanism to create distributed channel grid line objects (one for each channel on which lines are drawn).

DtChannelKeywordGridLines inherits from DtChannelKeywordObject, and depends on the GridLines channel keyword. When all the necessary conditions are met to draw grid lines (terrain loaded, 2D mode, grid lines enabled) DtChannelKeywordGridLine creates a DtChannelGridLines distributed object (which in turn creates a DtOsgChannelGridLinesInstance.

The standard DialogPage pattern is used to provide a page for setting grid lines settings (*DtGridLinesPage*, *DtGridLinesLogic*, *DtGridLinesInterface*).

The standard archival methodology is use to store and back up settings (a *DtGridLinesSettingsRecord* is created, and settings are read from and written to *default_GridLinesSettings.grdx*).

3D Graphics

This chapter describes the technologies VR-Vantage uses to render 3D graphics.

7.1. 3D Rendering

VR-Vantage uses several technologies for its visualization capabilities. The lowest common denominator is OpenGL. Layered on top of OpenGL is OpenSceneGraph (OSG). VR-Vantage builds new technologies and adapts several third party technologies on top of OSG.

7.1.1. OpenGL

OpenGL is an industry standard environment for developing 2D and 3D graphics applications. OpenGL is available on many different platforms and is actively supported by the leading graphics vendors. In most cases OpenGL is an API for controlling the functions of a graphics rendering architecture. While this architecture can be implemented using software executed by the CPU, it is more common that dedicated graphics hardware is used to accelerate these functions. VR-Vantage takes advantage of the latest capabilities of modern graphics hardware and they are required by most rendering features. OpenGL is a functional API specification typically used by the C language and languages which can interoperate with C.

7.1.2. OpenSceneGraph

OpenSceneGraph is a C++ scene graph API built on top of OpenGL. A scene graph is a data structure that represents a scene through connected nodes. OSG follows a number of common patterns for interoperating with the scene. One example is the visitor pattern in which an object is written in a way that allows it to visit each node in a graph and perform an operation such as culling. OSG provides many features that VR-Vantage can support, including a wide range of file format loaders, particle effects, weather, data management, paging data, and LOD management. If VR-Vantage does not support a particular OSG feature, it can be integrated with minimal effort.

OSG provides plug-ins for additional capabilities and file loading. VR-Vantage provides plug-ins for loading MÄK GDB files, MÄK propriety geometry (MEDF), and imagery (MEIF) files.

7.1.3. Rendering Images

The component responsible for drawing 3D images onto a screen is called a renderer. Each display engine owns one. VR-Vantage's renderer is based on OpenSceneGraph (OSG).

The scene graph starts with a top level root node. Beneath the root node, group nodes organize geometry, transforms, and state. Figure 7-1 represents an abstract scene graph.

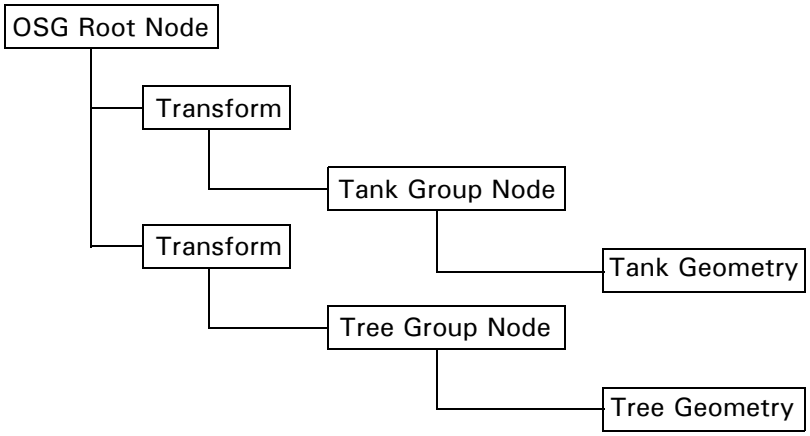


Figure 7-1. Scene graph

Distributed objects usually affect the scene graph. They can add nodes, remove nodes, change transforms, and change state. Therefore, when a master display engine calls an agent, it is causing changes to the scene graphs of external display engines. In Figure 7-2, the master display engine has an agent. The agent has a corresponding distributed object that exists in each display engine. The distributed object adds nodes to the scene (the Tank Group Node) and positions them in the world using a transform.

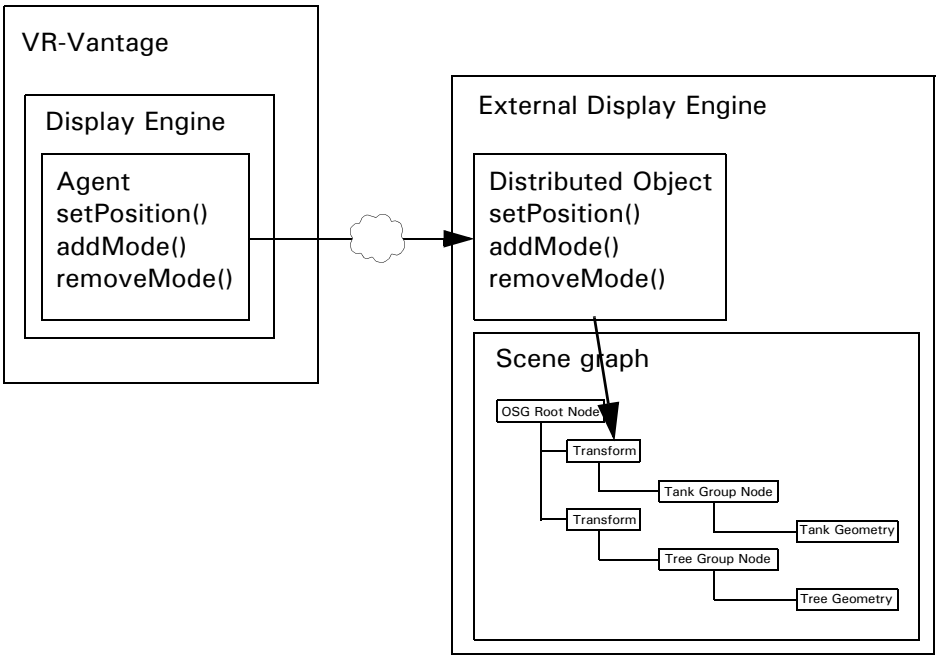


Figure 7-2. Agent and distributed object

7.2. The Renderer

The core VR-Vantage Toolkit framework abstracts away from the underlying rendering implementation. This separation ensures that the rendering-independent code remains independent of OSG, while the rendering code exists in the libraries that depend on OSG. (VR-Vantage can support alternate rendering solutions.) The class responsible for rendering in VR-Vantage is ***DtRenderer***. The *DtRenderer* provides an interface for updating the scene graph and rendering the scene. The *DtRenderer* instance is owned by the display engine. Its update() and render() functions are called each display engine tick.

7.2.1. The OSG-Specific Renderer

The *DtOsgRenderer* implements the *DtRenderer* interface and provides access to the various OSG nodes, such as the scene root node, and other nodes where various models are added. VR-Vantage provides classes for easily managing objects in the scene. The *DtOsgSceneConnector* manages transformation nodes in the scene graph.

The renderer also starts the OSG update traversal for updating all nodes in the scene. After the update traversal is complete, the renderer starts the render traversal, which causes the scene to be rendered on all the available channels.

7.3. Model Classes

In most cases it is not necessary to add new model classes. The *DtOsgSimpleModelInstance* class can load and display any file format supported by OSG and will automatically cache, accelerate, compress, and instance the rendering resources. However, if you need custom visualization, it is easy to add new model classes, and add instances that use the model classes to objects in the scene.

If you create custom model classes, you use them by creating a schema and model definition for the new model. Then you map it to an entity type for display. Please see VR-Vantage Users Guide for details.

7.4. OpenSceneGraph Extensions in VR-Vantage

VR-Vantage implements extensions to OpenSceneGraph. The SilverLining cloud and ephemeris model is integrated by a custom group node and two drawables. An OSG drawable is the generic interface that OSG provides for doing OpenGL drawing. The SpeedTree vegetation rendering is an example of a custom drawable. The *DtSpeedTreeDrawable* can make the SpeedTree rendering API calls. Once a custom model is created, it is free to create custom drawables, which are added to geode nodes.

7.5. Cockpit Display Updaters

A cockpit display updater is a C++ object that updates part of a GL Studio cockpit display (HUD). An updater examines the entity state and sends the cockpit display object or its agent the appropriate message. All updaters send their values as text strings, as required by GL Studio cockpit displays.

VR-Vantage provides updaters for the cockpit displays it ships. However, you can use the VR-Vantage Toolkit and GL Studio to replace updaters or add custom updaters to be used with the supplied cockpit displays or your own cockpit displays. Custom updaters must derive from ***DtGlStudioVrlAttributeUpdater*** or ***DtGlStudioObjectAttributeUpdater***. Updaters are created using a creator object. New updater creator objects are registered with the factory.

Creators have a name and a protocol. This is because updaters that work with DIS entities do not work with HLA entities. You can generate multiple updaters by using the shared source technique that is used elsewhere in the VR-Vantage Toolkit. (For more information, please see "Using Shared Source Files for Simulation Drivers," on page 5-8.) The supported protocols are DtDis, DtHla13, DtHla1516, and Object. The updater factory is in the ***DtSharedCockpitSettings*** object. Registering a new creator is done through its registerUpdater() function.

VR-Vantage provides the following cockpit display updaters:

- Object Protocol:
  - AOA-Angle of Attack. Returns the pitch of the entity clamped (-5 to 30 degrees).
  - FeetAGL-Feet Above Ground Level. Returns the feet above the terrain (or buildings below).
  - FeetMSL-Feet Above Mean Sea Level. Returns the altitude above sea level.
  - FeetPerMinute. Returns the climb rate.
  - KilometersPerHour. Kilometer per hour.
  - MilesPerHour. Miles per hour.
  - KnotsPerHour. Nautical Miles per hour.
  - Pitch. Returns the pitch of the entity clamped (-180 to 180 degrees).
  - Roll. Returns the roll of the entity clamped (-180 to 180 degrees).
  - Yaw. Returns the yaw of the entity clamped (0 to 360 degrees).
  - Value. This is a special updater in that it will send any value to the cockpit display once and then never again. It's used to configure initial setup and to turn display components on or off.
- DtDis, DtHla13, DtHla1516 Protocol:
  - Headlight. Returns the headlight state of the entity as true or false

Updaters can exist in the simulation thread, owned by the GL Studio Visualizer, or in the render thread, owned by the GL Studio cockpit display object.

Updaters that live in the simulation thread have access to the entire simulation state in the entity state repository. These updaters are updated at the same rate as the network thread (approximately 15 times per second). The values that these updaters send are not smoothed in the cockpit display. These updaters are derived from the *DtGlStudioVrlAttributeUpdater* and have a protocol of DtDis, DtHla13, or DtHla1516.

Updaters that live in the render thread are owned by the GL Studio cockpit display object. They have access to the cockpit display object's smoother. This smoother is updated by the simulation thread and the position, velocity, acceleration, and rotation are smoothed between updates. The render thread updaters update their values approximately 60 times per second. These updaters are registered with a protocol of Object.

It is important to install new updaters after the display engine is initalized. Connect to the display engine object's post_initialzie signal to be notified when it is done initializing. Then install your updaters.

7.6. Making OpenGL Calls from OSG Drawables

It is possible to call OpenGL directly from an OSG drawable. It is very easy to corrupt the scene rendering when you do this. OSG maintains a copy of the OpenGL state in its own state object osg::State. If this state object and the actual OpenGL state are not synchronized, then the rendering will not be correct. If you must call OpenGL directly, you must undo any changes to the OpenGL state that you make. After your code is executed, the OpenGL state must be returned to the values it had before your code was executed. This will ensure that the OSG and OpenGL states are synchronized.

The following pseudo-code is an example of what must happen for all OpenGL states:

```
//For all states you want, check the existing OSG state.
If(osg_state != desired_state)
{
   OPENGL_setState(desired_state);
}
//Call render commands
OPENGL_renderCommands();
//For all states you changed, restore back to the OSG state.
If(osg_state != desired_state)
{
OPENGL_setState(osg_state);
}
```

8

2D Graphics

This chapter describes the technologies VR-Vantage uses to render 2D graphics.

8.1. 2D Rendering

An overlay is a 2D scene that is drawn on top of a 3D scene. The 2D scene can contain any number of the 2D primitives, which include: DtLineSegment, DtQuad, DtFilled-Polygon, and DtOutlinedPolygon. The class that implements the overlay concept is the ***DtOverlayRenderer***. The *DtOverlayRenderer* does the actual drawing of the graphical primitives.

To draw an overlay you must add a ***DtOverlayDrawable*** to the scene. The *DtOverlay-Drawable* is responsible for telling overlays (instances of *DtOverlayRenderer*) to render themselves at the appropriate time.

The *DtOverlayDrawable* is told which *DtOverlayRenderer* it should draw. During the render phase, the *DtOverlayDrawable* tells the *DtOverlayRenderer* to render itself and the graphics primitives are then rendered.

The **exampleLogo** example shows how to create an overlay.

VR-Vantage Design Patterns

This chapter describes the design patterns that underlie much of the VR-Vantage API.

9.1. Design Patterns

The VR-Vantage Toolkit uses design patterns, an object-oriented programming technique described in *Design Patterns: Elements of Reusable Object-Oriented Software*[1].

9.2. The Rooted Singleton Pattern

The Singleton pattern provides a single global instance of a class to an application. The Rooted Singleton pattern provides a single global instance of a class associated with a specific root object. It allows Toolkit users to create applications that have two display engines without having them interfere with each other.

The Rooted Singleton pattern involves two classes, the singleton class and the root object class. The singleton class is globally accessed. The root object class is the scope in which the singleton class lives.

Suppose that you want to have two display engine instances in your application. You want to override one of them so that it creates your own derived channel whenever a channel is created, but you want the other display engine to behave normally. If we used the standard Singleton pattern you could not do that. If you overrode the channel creator with your own channel creator, then both display engines would create your derived channel, which is not what you want. The Rooted Singleton pattern lets you register a new channel creator with one instance of the display engine, but leaves the other instance alone.

To create a Rooted Singleton, do something like this:

```
// Example Rooted Singleton
class MyExampleSingleton : public DtVirtualBaseClass
{
public:

  // Lazily create a global instance rooted off a Display Engine
  static MyExampleSingleton& instance(DtDE displayEngine)
  {
    MyExampleSingleton* singleton =
      dynamic_cast<MyExampleSingleton*>(
      displayEngine.findInstance("MyExampleSingleton"));
    if(!singleton)
    {
      singleton = new MyExampleSingleton();
      displayEngine.registerInstance(
        "MyExampleSingleton",singleton);
    }
    return *singleton;
  }

  // DTOR
  virtual ~MyExampleSingleton();
```

1. Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley Professional, 1994.

```
    // Your methods here...

    void myExampleMethod()
    {
    }

  private:

    // Private CTOR.
    MyExampleSingleton();

  };
```

The singleton lazily creates and registers itself with the display engine. Under the hood, the display engine simply has a map of *DtVirtualBaseClasses* with std::string keys. When you access this object you use the static instance() method:

```
MyExampleSingleton::instance(displayEngine).myExampleMethod();
```

The first time this object is accessed, it is created and registered with the display engine using the key MyExampleSingleton. The next time it is accessed, the singleton is just returned. The singleton lives throughout the existence of the display engine. When the display engine is destroyed, all the Rooted Singletons are deleted.

9.2.1. The Rooted Singleton Pattern Specification

The Rooted Singleton pattern has the following requirements and behavior:

- The Rooted Singleton pattern involves two classes: the singleton class, of which there will be one globally accessible instance, and the root object class, which is the scope that the singleton exists in.
- The singleton class should not have a public constructor.
- The singleton class should provide a static instance accessor to easily get the singleton instance from the root object.
- The singleton class must be explicitly registered with the root object. This is usually done through lazy creation.
- The lifetime of a Rooted Singleton is linked to the lifetime of the root object.

Many of the other design patterns (Signaler, Creator, and Factory) use the Rooted Singleton pattern.

9.3. The Signaler Pattern

The Signaler pattern lets signals live outside the object about which they are signaling.

Sometimes signals for objects are hard to use. For example, if FirstObject wants to connect to a signal emitted by SecondObject, but SecondObject does not exist yet, creating the connection is a problem. We have created the Signaler pattern to solve this problem. Now, FirstObject connects to signals emitted by *SecondObjectSignaler*, a Rooted Singleton. Then SecondObject has the SecondObjectSignal emit the signals causing FirstObject to get signalled.

Here is a simple example:

```
// Class that wants to let others know about a change in speed.
class Foo
{
public:

  void setSpeed(double s)
  {
    mySpeed = s;
    FooSignaler::instance(
      myDisplayEngine).signal_speedChanged(this,mySpeed);
  }

protected:

  DtDE& myDisplayEngine;

};

// Class that emits signal_speedChanged;
class FooSignaler
{

public:

  // Lazily create a global instance rooted off a Display Engine
  static FooSignaler& instance(DtDE displayEngine)
  {
    FooSignaler* singleton = dynamic_cast<FooSignaler*>(
      displayEngine.findInstance("FooSignaler"));
    if(!singleton)
    {
      singleton = new FooSignaler();
      displayEngine.registerInstance("FooSignaler",singleton);
    }
    return *singleton;
  }

  // Signal to emit.
  signal < void (Foo*, double) >  signal_speedChanged;
};
```

Now, an interested party can register with the *FooSignaler* before an instance of the Foo object actually exists:

```
FooSignaler::instance(myDisplayEngine).signal_speedChanged.connect(
  boost::bind(&InterestedPartyClass::onSpeedChanged,
```

```
    &insterestedPartyClassInstance, _1)));
```

The signaler is lazily created and the *InterestedPartyClass* gets signaled whenever an instance of Foo comes into existence and setSpeed() is called on it.

9.3.1. The Signaler Pattern Specification

The Signaler pattern has the following requirements and behavior:

- The Signaler pattern involves two classes: the subject class, which wants to let others know when something has changed, and a signaler class, the object that parties interested in updates register with.
- The signaler class should not have a public constructor.
- The signaler class should provide a static instance accessor to easily get the signaler instance from the root object.
- The signaler class must be explicitly registered with the root object. This is usually done through lazy creation.
- The lifetime of a signaler is linked to the lifetime of the root object.
- The subject class uses the signaler class to emit signals when things in the subject class change.
- Interested parties connect to the signals in the signaler to find out about changes in the subject class.

9.4. The Factory Pattern

The Factory pattern allows creation of specializations on a single base class within the scope of a root object.

The Abstract Factory pattern creates an instance of an object from a family of objects without having to specify the concrete class to be created. Our Factory pattern extends this to include Rooted Singleton support, meaning each root object (display engine) will have its own factory and there can be multiple root objects in an application.

VR-Vantage uses the Factory pattern a lot. From creating different kinds of windows, to creating different types of file parsers, it is an important pattern to understand.

Here is an example Factory pattern:

Suppose your program needs to be able to parse different types of terrain files and determine the coordinate system of each one.

1. Write the base class header parser. This provides the API that all your derived file format parsers would conform to. For example, you will need an OpenFlight parser, a DTED parser, 3DS parser, and so on.

```
// Abstract file parser class.
class ExampleFileParser
{

public:

  // CTOR
  ExampleFileParser(DtDE& displayEngine, const std::string& filePath);

  // DTOR.
  virtual ~ExampleFileParser();

  // Returns the terrain coordinate system.
  virtual const std::string& terrainCoordinateSystem() = 0;

protected:

  DtDe& myIg;
  std::string myFilePath;
  std::string myTerrainCoordinateSystem;

};
```

2. Create the file parser factory:

```
class DtFileParserFactory : public DtVirtualBaseClass
{
  public:

  // Map of file extensions to creators.
  typedef std::map<std::string, ExampleFileParserCreator*> CreatorMap;

  // Get the instance of the factory.
  static ExampleFileParserFactory& instance(DtDE& displayEngine)
  {
    ExampleFileParserFactory* factory =
```

```
    dynamic_cast<ExampleFileParserFactory*>(
    displayEngine.findInstance("ExampleFileParserFactory"));
  if(!factory)
  {
    factory = new ExampleFileParserFactory(displayEngine);
    displayEngine.registerInstance(
      "ExampleFileParserFactory",factory);
  }
  return *factory;
}

// DTOR
virtual ~ExampleFileParserFactory()
{
  DtClearAndDeletePointersInMap(myCreators);
}

// Add a creator based on file extension
virtual void addCreator(const std::string& extenstion,
  ExampleFileParserCreator* creator)
{
  CreatorMap::iterator iter = myCreators.find(extenstion);
  if(iter != myCreators.end())
  {
    delete iter->second;
    iter->second = creator;
  }
  else
  {
    myCreators[extenstion] = creator;
  }
}

// Create an ExampleFileParser.
ExampleFileParser* createFileParser(const std::string& filePath)
{
  std::string extenstion = fileExtensionFromFilePath(filePath);
  CreatorMap::iterator iter = myCreators.find(extenstion);
  if(iter != myCreators.end())
  {
    return iter->second->create(filePath);
  }
  else
  {
    DtTHROW_NEW(
      DtInvalidInput,"No creator found with the given name.");
  }
}

// Get the creators.
const CreatorMap& creators() const
{
  return myCreators;
}

protected:

// CTOR
ExampleFileParserFactory(DtDE& dislayEngine);
```

```
  DtDE& myDisplayEngine;
  CreatorMap myCreators;

};
```

3. There is also the base class file parser creator class:

```
// Abstact file parser creator class.
class ExampleFileParserCreator
{
  public:

    // CTOR.
    ExampleFileParserCreator();

    // DTOR.
    virtual ~ExampleFileParserCreator();

    // Creator method.
    virtual ExampleFileParser* create(DtDe& ig,
    const std::string& filePath) const = 0;

};
```

4. Derive the file format parsers and their creators, and register them with the creator:

```
// Openflight header parser class.
class DtOpenFlightFileParser : public ExampleFileParser
{

public:

  // CTOR
  ExampleOpenFlightFileParser(DtDE& displayEngine,
  const std::string& filePath);

  // DTOR.
  virtual ~ExampleOpenFlightFileParser();

  // Returns the terrain coordinate system.
  virtual const std::string& terrainCoordinateSystem()
  {
    // open file. parse for coordinate system.
    return myTerrainCoordinateSystem;
  }

};

// ExampleOpenFlightFileParserCreator class.
class ExampleOpenFlightFileParserCreator : public
  ExampleFileParserCreator
{
  public:

  // CTOR
  ExampleOpenFlightFileParserCreator();

  // DTOR
  virtual ~ExampleOpenFlightFileParserCreator();

  // Create a header parser
```

```
  virtual ExampleFileParser* create(DtDE& displayEngine,
    const std::string& filePath) const
  {
    return new ExampleOpenFlightFileParser(displayEngine, filePath);
  }

};

ExampleFileParserFactory::instance(myDisplayEngine).addCreator(
  "flt", new ExampleOpenFlightFileParserCreator);

// 3DS header parser class.
class Dt3dsFileParser : public ExampleFileParser
{

  public:

  // CTOR
  Example3dsFileParser(
    DtDE& displayEngine, const std::string& filePath);

  // DTOR.
  virtual ~Example3dsFileParser();

  // Returns the terrain coordinate system.
  virtual const std::string& terrainCoordinateSystem()
  {
    // open file. parse for coordinate system.
    return myTerrainCoordinateSystem;
  }

};

// Example3dsFileParserCreator class.
class  Example3dsFileParserCreator : public ExampleFileParserCreator
{
  public:

    // CTOR
    Example3dsFileParserCreator();

    // DTOR
    virtual ~Example3dsFileParserCreator();

    // Create a header parser
    virtual ExampleFileParser* create(DtDE& displayEngine,
    const std::string& filePath) const
    {
      return new Example3dsFileParserCreator(displayEngine, filePath);
    }

};

ExampleFileParserFactory::instance(myDisplayEngine).addCreator(
   "3ds", new Example3dsFileParserCreator);
```

9.4.1. The Factory Pattern Specification

The Factory pattern has the following requirements and behavior:

- Factories are not subclassed. The factory only affects the base class.
- Factories can be lazily created (that is, created on first access).
- A factory's lifetime is linked to the root object it is registered with. Factories are Rooted Singletons.
- Factories must create new objects and can only create new objects. They should not be used to access existing objects (for that see the provider pattern).
- Factories provide a way to register a creator functionality. Creator functionality may be a function, functor, or creator-like object.
- Factories must have a protected CTOR. Instances are accessible through a static instance function (which does lazy registration).
- Factories do not have to be explicitly registered.

9.5. The Creator Pattern

The Creator pattern lets you replace a single class in a system with a specialized subclass. You might do this to fix a bug, or to add or change the behavior of specific functions. This is usually done through the Factory Method pattern. Unfortunately, if an application allows multiple parallel hierarchies (such as two display engines in the same application), the Factory Method pattern fails, because it indiscriminately sets the override class for all users. To implement similar behavior, but allow for changes to a class implementation tied to a specific root object, VR-Vantage uses the Creator pattern.

The Creator pattern involves two classes, the class which can be replaced (called the object class) and a class that does the creation (called the creator class). The creator instance is a Rooted Singleton of a given type registered with the root display engine. It can be accessed by anyone who has a root object pointer. All users of the object class use a Rooted Singleton of the creator class to create an instance of the object class.

The **exampleCreator** example uses the Creator pattern to cause a derived channel to get created when the display engine creates a channel. For details, please see the exampleCreator example in the class documentation.

9.5.1. The Creator Pattern Specification

The Creator pattern has the following requirements and behavior:

- The Creator pattern involves two classes, the class that can be replaced (called the object class), and a class that does the creation (called the creator class).
- All users of the object class use a 'pseudo-global' instance of the creator class to create an instance of the object class. The creator instance is pseudo-global because a single instance of a given type is registered with the root object and can be accessed by anyone who has a root object pointer (in this case the root object is the display engine object).
- The object class should not have a public constructor.
- The object class should use mostly virtual functions.
- The object class should specify the creator as a friend class.
- The creator class should provide a virtual create function.
- The creator class should provide a static creator instance accessor to easily get the creator instance from the root object.
- The creator instance must be explicitly registered with the root object before anyone attempts to create a class object.
- The creators should be registered during initialization.
- The lifetime of a creator is linked to the lifetime of the root object.
- To override the object class, sub-class the creator and implement the virtual create function to return a new subclass instance.

9.6. The Humble Dialog Pattern

Developers often put a lot of logic code inside their dialog boxes. This makes it hard to change the logic without changing the dialog box, hard to change the dialog box without changing the logic, and makes it impossible to test the logic outside of the GUI (say in an automated tester).

VR-Vantage uses the Humble Dialog pattern to solve this problem. The Humble Dialog pattern provides a smart, tested logic class, and a minimalistic dialog box class.

Figure 9-1 illustrates a *Logic* class that does all the work. Its constructor takes an interface class pointer. The interface is an abstract base class that has methods the logic calls to cause changes in the widget.

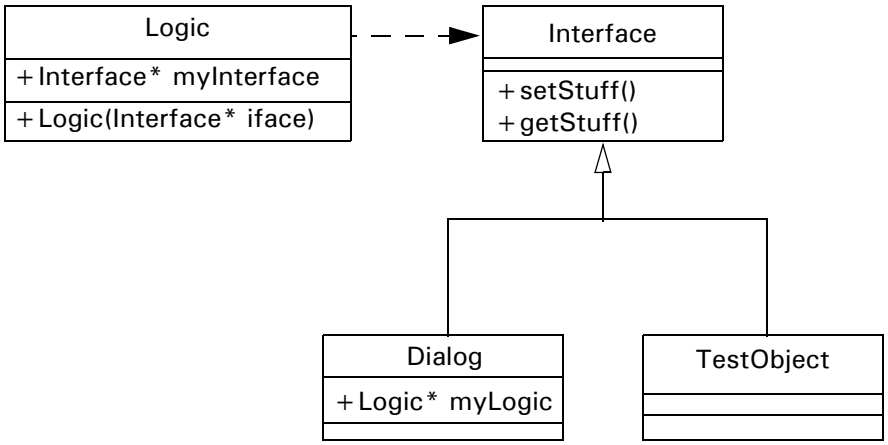


Figure 9-1. Humble dialog pattern

The dialog class implements the interface and creates an instance of the *Logic* class. The TestObject also implements the interface class, but does not need to have any of the GUI elements.

The following example shows how to use this pattern. Suppose you want to write a dialog box that lets the user convert a position in one coordinate system into another coordinate system.

First, you would write the logic class. It might look something like this:

```
class ConverterLogic
{
public:

  ConverterLogic(ConverterInterface* iface);

  // Convert position1 into a new coordinate system and
  //store result in position2.
  bool getConvertPostion(const Point3d& postion1, Point3d& position2)
  {
    // do the conversion. Now, position2 has the new point...
```

```
    // makes a call onto the dialog via the Interface pointer.
    myInterface->setPositionWidget(position2);
  }

};
```

The logic depends on the interface class to provide the methods it can call:

```
class ConverterInterface
{
public:

  void setPostionWidget() = 0;
};
```

The dialog class implements the interface:

```
class ConverterDialog : public ConverterInterface
{
public:

  ConverterDialog()
  {
    // Create the logic and give it a pointer to this dialog.
    myLogic = new ConverterLogic(this);
  }

  // Make the call to change this dialogs position widget.
  void setPositionWidget(const Point3d& position);

protected:

  ConverterLogic* myLogic;

};
```

The dialog creates the logic using itself as the implementation of the interface.

You could write a *ConverterTester* class to test the *ConverterLogic* without needing to run an application with a GUI. This is useful for automated testing.

```
class ConverterTester : public ConverterInterface
{
public:

  void setPostionWidget(const Point3d& position)
  {
    if(position != correctPosition)
    {
      testFailed = true;
    }
  }
};
```

9.6.1. The Humble Dialog Pattern Specification

The Humble Dialog pattern has the following requirements:

- A logic class that does all the work.
- A dialog or widget interface class that the logic will talk to.
- A widget class that implements the interface.
- Optionally, a test object to test your logic.

10

Building VR-Vantage Applications

This chapter describes API conventions and how to build and compile applications.

10.1. VR-Vantage Toolkit Libraries

VR-Vantage libraries are organized by dependencies. The VR-Vantage Toolkit uses strict naming conventions to indicate the dependency of the library. For example, *vrvVrl* is the VR-Link protocol-independent simulation library. It provides simulation specific but protocol-independent classes. The *vrvVrlQt* library provides simulation specific protocol-independent GUI classes. The *vrvVrlQt* library depends on *vrvVrl*. If you are developing an application that uses these classes and uses Qt, you can use both *vrvVrl* and *vrvVrlQt* libraries. The plug-in libraries are very small libraries, that usually have one or two functions with C language symbols. The plug-in libraries allow for runtime loading and initialization of VR-Vantage libraries. VR-Vantage applications are built using all VR-Vantage libraries as plug-ins, for maximum flexibility and extensibility. *vrvVrlPlugin* is the plug-in library for the *vrvVrl* library. There is a corresponding *vrvVrlQtPlugin* library for the *vrvVrlQt* library.

For a list of libraries and details about what they do, please see the class documentation.

10.2. Conventions

This section describes the conventions used in the VR-Vantage Toolkit for including files and using namespaces.

10.2.1. Include Files

All header files include the directory path relative to the *include* directory. For example, the file *./include/makVrv/DtDe.h* is included using the include line

```
#include <makVrv/DtDe.h>
```

The forward slash is used to ensure cross-platform compatibility. The convention is also followed by OpenSceneGraph and Boost.

10.2.2. Namespaces

All VR-Vantage Toolkit classes are in the makVrv namespace. When referring to a VR-Vantage Toolkit class in a header file, you must specify the namespace.

To predeclare a class in a header file, use the following syntax:

```
namespace makVrv
{
  Class DtDe;
}
```

To use a class in a header file, use the following syntax:

```
class Test
{
   void doSomething(makVrv::DtDe& de);
};
```

When writing code in a source file, either specify the namespace with the type or use the `using namespace` declaration, as follows:

```
void Test::doSomething(makVrv::DtDe& de) { … }
```

or:

```
using namespace makVrv;
void Test::doSomething(DtDe& de) { … }
```


Never use the `using namespace` declaration in header files. This will leak the namespace into all files that use that header and potentially cause irresolvable naming conflicts.

Most libraries have a secondary namespace that is used when there are duplicate names, such as the standard init() functions, or in the protocol-specific libraries when they share the same source code. The name of the secondary namespace matches the name of the library.

For example, *vrvCoreQt* has the secondary namespace *vrvCoreQt*, which has the library's init() function. To initialize the *vrvCoreQt* library, use the following code:

```
void init(makVrv::DtDe& de)
{
  // Initialize the Qt library.
  makVrv::vrvCoreQt::init(de);
}
```

You must initialize *vrvCoreQt* if you are building an application that links to it or a plug-in that depends on it.

10.2.3. Naming Conventions

The VR-Vantage Toolkit follows the standard MÄK policy of prefixing all classes with "Dt".

All functions, variables, and enumerations are inside classes. Subtypes and enumerations do not get a leading Dt. All class member variables start with the word "my". All class static variables start with the word "the".

The VR-Vantage Toolkit uses the following conventions for naming functions:

- Functions that set the internal state of an object start with the word "set", for example:
  ```
  void setSimulationTime(double);
  ```
- Functions that return a value through a passed reference start with the word "get":
  ```
  void getLocation( double &x, double &y, double& z ) const;
  ```
- Functions that return a valid state value in the return type use the name of the value without any qualifying verb, for example:
  ```
  double simulationTime() const;
  ```
- Functions that return an object that may not be valid start with the word "find", for example:
  ```
  DtChannel* findChannel(const std::string&);
  ```
  The use of find is to encourage testing the validity of a variable that might be invalid. Functions that return potentially invalid variables through references and return a boolean can also use the word find.
- Functions that delete or destroy an object contain the word "destroy", for example:
  ```
  destroyModelDefinition( const DtModelDefinition&)
  ```
- Functions that remove an object from another object's internal management contain the word "remove", for example:
  ```
  removeModel(makVrv::DtModelDriver*);
  ```
  This may assume a transfer of ownership if the second object was the sole manager of the first object.
- Most signals that are based on Boost start with the prefix "signal_". Signals that occur before an event contain the words ToBe and then the event, for example:
  ```
  signal_terrainPatchToBeDeleted
  ```

There are some intentional exceptions to these conventions, usually to improve clarity of the code.

10.3. The Qt Toolkit

VR-Vantage uses the Qt™ Toolkit from Qt Software to build the graphical user interface (GUI). This section provides general information about the Qt toolkit and Qt Designer. For more information, please see Qt Software documentation.

10.3.1. Qt Designer

Qt Designer is a Qt application for designing and building user interfaces interactively. In Qt Designer, you can create custom widgets and save the results to a file. The files saved by Qt Designer (*.ui* files) are called forms. These files are XML files that you compile to source code using the Qt User Interface Compiler (UIC). The UIC takes a form file as input, and generates C++ header and source files as output. The resulting source files can then be compiled and linked into an application.

10.3.2. Qt Signals and Slots

Qt provides a mechanism for communicating between objects, known as signals and slots. Under this scheme, an object can emit a signal representing some event, and any number of objects can receive the signal if they are connected to the signal through slots (functions to handle the signal). It is a flexible alternative to traditional callback member functions. The Toolkit uses the Qt signal/slot mechanism for communicating between objects. If you plan to use this capability in your derived classes, remember the following:

- Only those classes derived from the Qt class *QObject* can use signals and slots.
- You must include the Q_OBJECT macro in the class definition of derived classes that contain signals and slots. Based on Qt documentation, we recommend that you include it in all classes derived from *QObject*.

10.4. Boost Notes

This section describes issues specific to the use of the Boost library. For complete Boost documentation, please got to its web site: http://www.boost.org/

10.4.1. NOMINMAX (Windows Only)

The latest versions of Boost include *Windows.h*, which redefines the min and max macros. The macros cause compile time problems with the standard library implementation. The temporary solution is to disable *Windows.h* and redefine the macros using the NOMINMAX macro. This is typically done in the Preprocessor settings for all projects.

10.4.2. Boost and Qt Signals

Signals and slots are a way of communicating between objects. An object can emit a signal representing some event and any number of objects can receive the signal. Boost and Qt both use signals. The VR-Vantage Toolkit uses Qt signals in the GUI and Boost signals everyplace else.

Unfortunately Qt defines a macro that conflicts with the Boost signals namespace. Boost has a workaround in which it creates a namespace alias. VR-Vantage provides a header, ***signalslib.h***, that implements this workaround. To use it, include *signalslib.h* in any file that uses both Qt and Boost signals before any Qt header. *vrvCoreQt* and all other VR-Vantage Qt libraries do this automatically, so most developers will not need to do this.

10.5. Compiling and Linking the VR-Vantage Applications

Before you extend the VR-Vantage applications, write plug-ins, or develop new applications using the VR-Vantage Toolkit, please review the licensing issues covered in "Third-Party Software and Content," on page 1-3.

10.5.1. Windows

Development in Windows is usually done using Microsoft Visual Studio. The example projects and the application source include project files to get you started.

Add the root *include* directory to the include directory list for the project:

```
vrvantage\include
```

All VR-Vantage Toolkit libraries are automatically linked, if necessary. No explicit linking is required. Only the *lib* directory needs to be added to the library directory list, as follows:

```
vrvantage\lib
```

You must explicitly link VR-Link, OpenSceneGraph, and any other libraries that do not perform auto-linking.

10.5.2. Linux

To compile using gcc, you must add several parts to the compile line.

To compile a source file to an object, add the include directories:

```
-I vrvDir/include –I $(MAK_VRLDIR)/include
```

To link an application or shared library, add the linking directories:

```
-L vrvDir/lib –L $(MAK_VRLDIR)/lib
```

Add any other libraries that were used, for example:

```
-lvrvCore –lvrvUtil -lvlutil
```

You must list libraries in order of decreasing dependency. Since *vrvUtil* depends on *vlutil* it must come first in the list.

When compiling with VR-Link protocol-specific libraries, the normal VR-Link compiling rules apply.

10.5.3. Building Examples

The VR-Vantage Toolkit includes examples of plug-ins and VR-Vantage-based applications.

To build the examples:

1. Define the following environment variables:
   - QTDIR. Directory of your Qt installation
   - MAK_VRLDIR. Directory of your VR-Link installation
   - MAK_RTIDIR. Directory of your RTI installation (if you are building for HLA)
   - MAK_CIGIDIR. Directory of your CIGI installation (if you are building a CIGI example).
2. In Windows, open *./examples/vrvantageExamples.sln* and compile the examples. In Linux, run 'make' from the directory of the example you want to build.

For information about the individual examples, please see the Examples link in the class documentation.

10.6. Distributing Data with Plug-Ins

If you write a plug-in that affects schemas, model definitions, or element definitions, you might have new or modified settings that you want to install with your plug-in. You would have created the modified settings file by adding and editing the appropriate settings in VR-Vantage and saving the settings to file.

You could install your settings files and ask your customer to merge in the settings by hand in the GUI, or you can merge them in as part of your installation process. VR-Vantage includes a tool, vrvMerge, for merging settings.

The merge tool can merge files that have the following extensions:

- SMTX – schema definition file. The schema definition files provided with VR-Vantage are:
  - *Model Definitions.smtx*
  - *Visual Definitions.smtx*.
- OMMX – model definition file. VR-Vantage has one model definition file – *Model Definitions.ommx*.
- VODI – visual definition file. The visual definitions files provided with VR-Vantage are:
  - *EntityElementDefinitions.vodi*
  - *InteractionsElementDefinitions.vodi*
  - *TacticalGraphicsElementDefinitions.vodi*.

vrvMerge has the following syntax:

```
INSTALL_DIR/bin/vrvMergeTool input_file merge_file
  [output_file]
```

where:

- INSTALL_DIR is an environment variable that specifies the VR-Vantage installation directory. (VR-Vantage does not add itself to the system path, so the command must specify a fully qualified path to vrvMerge.)
- `input_file` specifies the settings file to merge into VR-Vantage.
- `merge_file` is the settings file to update.
- `output_file` is an alternative output file for the merged settings. By default, output is to the file specified by `merge_file`.

If the file extensions do not match or if an extension is not supported, vrvMerge prints an error.

The following example merges MyModelDefinitions into the default model definitions file.

```
[INSTALL_DIR]/bin/vrvMergeTool "MyModelDefinitions.ommx"
  "Model Definitions.ommx"
```

You can run this tool as many times as you wish for model definitions.

The merge tool updates the settings files in *./appData*. If a user restores the factory settings, the merged settings will be lost. You will need to provide your customer with instructions for merging in your modified files if they get overridden.

11

The VR-Vantage Control Toolkit

The VR-Vantage Control Toolkit lets you control VR-Vantage applications from remote applications.

11.1. The VR-Vantage Control Toolkit

The VR-Vantage Control Toolkit (VCT) lets you send DIS or HLA messages that can remotely control a VR-Vantage application. It supports the VR-Vantage attach modes and most visual states, such as track histories, trailing effects, entity labels, and so on.

The control messages store all data in a network-centric format. For example, locations are geocentric and entity IDs are strings with the protocol specific global ID of the entity. The control messages are versioned so that future versions of the Toolkit will be backwards compatible with messages sent by applications or recordings that use this version.

All control messages are transported in the data interaction of the specific protocol. The *vrvControl* library provides a templated data marshaller class (***DtVrvControlMessageMarshaller_v0*** that is responsible for loading and unloading data interactions with the control message. The supported commands are in ***vrvControlToolkitDataTypes.h***. The exampleSendControl example is a simple application that sends a remote control message.

Effective with release of the VR-Vantage Control Toolkit, the Stealth Control Toolkit is deprecated. VR-Vantage will continue to support a limited set of Stealth Control PDUs. However, the Stealth Control Toolkit will no longer be provided with VR-Vantage.

11.1.1. Implementation Details

The VR-Vantage Control Toolkit installs protocol-specific accessories for each of the simulation connection drivers (DIS/HLA13/HLA1516). The accessory installs a data interaction callback with the exercise connection and a control processor with the driver. The data interaction callback receives data interactions from the exercise connection and uses the data marshaller to retrieve the VCT control message. The control message is then passed to the control processor.

When the control processor is created, it creates a command execution agent, which in turn creates a command execution object. When the control processor receives a VCT control message, it converts it to a VR-Vantage command through the VR-Vantage command factory. The command factory must be protocol-specific to do the conversion. This conversion process is required to convert geocentric coordinates to local database coordinates and to convert entity IDs to VR-Vantage-specific unique IDs. Once the control message is converted to a command, the command execution agent's execute command function is called. The command execution agent passes the command to command execution object through the standard VR-Vantage agent/object mechanism for execution in the main (display engine) thread.

11.1.2. Adding a New Remote Control Message

If you add a new feature to a VR-Vantage application, you may want to be able to control it using the VR-Vantage Control Toolkit, in which case, you must add a new control message to the toolkit and new command logic in the VR-Vantage application to receive the control message.

To add a control message to the VR-Vantage Control Toolkit:

1. Create a new enum value for identifying the new control in the enum DtVrvControlTypeEnum in ***vrvControlToolkitDataTypes.h***. Add the enum to the end of the list. Adding it anyplace else will break backward compatibility of other messages.
2. Create the new control struct. It must subclass ***DtVrvControl_v1*** (also in *vrvControlToolkitDataTypes.h*) and contain any parameters that are sent over the network. Keep in mind that this is a NetStruct, and parameters should not be stored as pointers, because they will not properly be transmitted over the network.
3. Add logic to your remote control application to send the message.

To add a new command to a VR-Vantage application:

1. Create the command for initializing and executing the control command:
   - Create a new class in *vrvCore* that subclasses ***DtVrvCommand*** and ***DtVrvCommandCreator***.
   - The command should provide accessors for the parameters and logic for executing the command in the execute(DtDe& de) method.
   - The creator should initialize the command with the parameters from the control message in the create(DtVrvControl_v1*) method.
   - For an example, please see *./include/vrvCore/DtFollowViewCommand.h* and *./include/vrvCore/DtFollowViewCommand.cxx*.
2. Register the command creator so that the command is properly initialized and created when the control comes in from the network. In your plugin's init(DtDe&) method, connect a method (postInitializeRegisterCommands()) to the de.signal_postInitialize signal. This method should disconnect from the signal (since we no longer need to be connected to it), and register the command with the ***DtSharedVrvCommandSettings*** by doing the following:

```
void postInitializeRegisterCommands( DtDe* de )
{
  de->signal_postInitialize.disconnect( boost::bind(
    &postInitializeRegisterCommands, &de ) );
  {
    DtWriteSettingsLock<DtSharedVrvCommandSettings> settings(
      DtSharedSettingsManager::instance(*de) );
    DtSharedVrvCommandSettings::DtVrvCommandSeedMap& seeds =
      settings->seedMap();
    seeds[vrvControlToolkit::Control_NewControlType] =
      new DtVrvCommandCreatorSeed<DtNewCommandCreator>();
  }
```

```
}
```

Where vrvControlToolkit::Control_NewControlType is the enumeration you created for the new control, and DtNewCommandCreator is the creator for the command you created to handle the control message.

11.2. The VR-Vantage Remote Draw API

The VR-Vantage Remote Draw API allows third party applications to create, send, and manage 2D and 3D objects to be drawn by VR-Vantage.

11.3. Major Classes

The VR-Vantage Remote Draw API uses three classes extensively when creating and modifying objects. ***DtVdcVector2D*** is a 2D vector class that is similar in functionality to the VR-Link *DtVector* class – without the need to set a Z-value to 0.0. This is used in all the 2D object classes.

DtVdcColor is used to specify colors in either RGBA (Red, Green, Blue, Alpha) or HSVA (Hue, Saturation, Value, Alpha) formats. *DtVdcColor* also defines const values for several of the most commonly used colors. ***DtVdc3DAttachPoint*** is only used by 3D Objects, and is discussed in "3D Objects," on page 11-13.

The following example creates a 2D position in the middle of the VR-Vantage window:

```
DtVdcVector2D position( 0.5, 0.5 );
```

The following example creates a color class and sets it to red:

```
DtVdcColor color( 255, 0, 0, 255 );
```

The following example creates a color class and sets it to red using a predefined constant:

```
DtVdcColor color = DtVdcColor::DtVdcRed;
```

The following example changes the color to blue using the Hue parameter:

```
color.setHue( 160 );
```

11.4. Objects

You can use the VR-Vantage Remote Draw API to draw several types of objects and text in 2D and 3D. You can specify the parameters of each type of object. Common default parameters are supplied for simplified use. In addition, there are accessors and mutators for each object parameter supported. Every object needs a unique ID. The ID is used to generate the global ID for the object. Each object must also take a pointer to the *DtExerciseConn* object.

The following example creates and displays a red circle in the center of the window.

```
int nextId=1;
DtVdcVector2D position( 0.5, 0.5 );
DtVdcVector2D size( 0.2, 0.2 );
DtEllipseObject* circle = new DtEllipseObject( exconn, nextId++,
  position, size );
circle->setColor( DtVdcColor::DtVdcRed );

circle->update();
```

To change the color of the circle from red to green, and tell VR-Vantage about the change, do the following:

```
circle->setColor( DtVdcColor::DtVdcGreen );
circle->update();
```

To remove and destroy the circle object, do this:

```
delete circle;
```

11.5. 2D Objects

The VR-Vantage Remote Draw API supports 2D objects such as lines, rectangles, ellipses, and triangles (Figure 11-1). It also supports text. As seen in the example in the previous section, positions and sizes are specified as fractions of the current window size in X and Y (or width and height) using the ***DtVdcVector2D*** class. You can specify a counter-clockwise rotation in radians.

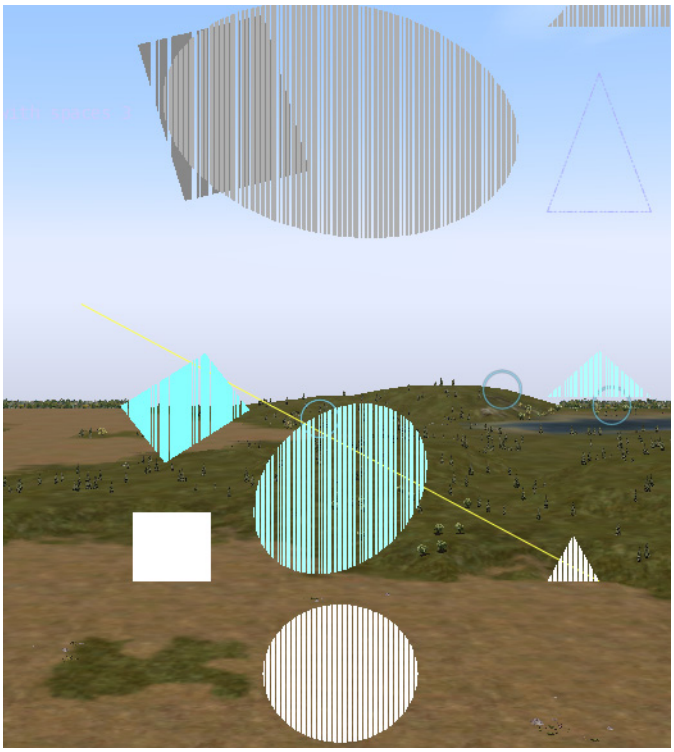


Figure 11-1. 2D ellipses

11.5.1. 2D Lines

DtLine2DObject represents a line that can be positioned and displayed on the screen. The line can be divided into multiple segments of varying relative lengths. Each line has a start point and an end point, specified using *DtVdcVector2Ds*. You can specify a color, line style, arrow style, pattern, and thickness for each segment, and the line can flash.

Line styles are specified using the DtVdcLineSegment::DtVdcLineStyle enumeration, which includes:

```
DtVdcLineSegment::DtVdcSegmentStyleStraight
DtVdcLineSegment::DtVdcSegmentStyleLightning.
```

Arrow styles are specified using the DtVdcLineSegment::DtVdcSegmentArrowStyle enumeration, which includes:

```
DtVdcLineSegment::DtVdcSegmentArrowStyleNoArrow
DtVdcLineSegment::DtVdcSegmentArrowStyleArrowEnd
DtVdcLineSegment::DtVdcSegmentArrowStyleArrowStart
DtVdcLineSegment::DtVdcSegmentArrowStyleArrowDouble.
```

The following example creates and displays a green line that goes from the bottom left-hand corner of the screen to the top right hand corner of the screen. The first 20 percent of the line should be a separate segment with double arrowheads:

```
DtLine2DObject* line = new DtLine2DObject( exConn, nextId++,
  DtVdcVector2D( 0., 0. ), DtVdcVector2D( 1., 1. ), 2 /*segments*/,
  DtVdcColor::DtVdcGreen );

line->setArrowStyle(
  DtVdcLineSegment::DtVdcSegmentArrowStyleArrowDouble,
  0 /* segment num */ );
```

11.5.2. Rectangles

DtRectangleObject represents a rectangular object that can be positioned and displayed on the screen. The position specified is the lower lefthand corner of the rectangle. The size parameter specifies the length of the x-axis sides and y-axis sides, respectively (as fractions of the window size). You can specify a counter-clockwise rotation, in radians, the rectangle's color, an optional fill pattern, and whether or not it flashes.

The following example creates and displays a white rectangle at the center of the screen. It is 1/10 the size of the screen along the x-axis and 1/5 of the screen along the y-axis:

```
DtRectangleObject* rect = new DtRectangleObject(exConn, nextId++,
  DtVdcVector2D( 0.5, 0.5 ), DtVdcVector2D( 0.1, 0.2) );
```

11.5.3. Ellipses

DtEllipseObject represents an elliptical object that can be positioned and displayed on the screen. The position specified is the center of the ellipse The size parameter specifies the length of the ellipsis' axes parallel to the screen's x-axis and y-axis. You can specify a counter-clockwise rotation, in radians, a color, and optionally, a fill pattern. The object can flash.

The following example creates and displays a flashing yellow circle in the lower right corner. The diameter is ½ the screen.

```
DtEllipseObject* circle = new DtEllipseObject(exConn, nextId++,
  DtVdcVector2D( 0.75, 0.25 ), DtVdcVector2D( 0.5, 0.5 ), 0.,
  DtVdcColor::DtVdcYellow);

circle->setFlashing( true );
```

To hide the ellipse, do this:

```
circle->setVisible(false);
```

11.5.4. 2D Triangles

DtTriangle2DObject represents a triangle that can be drawn on the VR-Vantage overlay. Each vertex is specified in fractional screen coordinates. Triangles can be rotated, scaled along the X and Y axes independently, and offset from their specified position. They can be colored, filled with an optional pattern, and can flash.

Because scaling is in screen coordinates, the effect of scaling will vary depending on the aspect ratio of the screen on which the graphic is viewed.

The following example creates and displays an orange-striped pattern-filled triangle that covers the upper right-hand side of the screen.

```
DtTriangle2DObject* triangle = new DtTriangle2DObject( exConn, nextId++,
  DtVdcVector2D( 0., 1. ), DtVdcVector2D( 1., 1. ), DtVdcVector2D( 1., 0.
  ) );

triangle->setFill( true );
triangle->setPattern( 0x0f0f0f0f );
triangle->setColor( DtVdcColor::DtVdcOrange );
```

The vertices of 2D triangles are specified as three points (v_1, v_2, v_3) and one offset. The position of each vertex before rotation and scaling is `v_n + offset`, and the rotation and scale operations are centered around the offset.

Figure 11-2 illustrates a triangle with its vertices specified as v_1 = (0, 0), v_2 = (0.5, 0), v_3 = (0, 0.5) and an offset of (0.5, 0.5). Applying the offset to the specified vertices, it is displayed with vertices at (0.5, 0.5), (1, 0.5), and (0.5, 1). (Remember vertices are specified as screen coordinates, with 0,0 being the lower left corner of the screen and 1,1 the upper right.)

If the triangle is rotated pi/2 (90^o), it rotates around the origin, as shown in the second view. If it is rotated and scaled 50%, it is displayed with vertices at (0.5, 0.5), (0.75, 0.5), (0.5, 0.75), as shown in the third view.

The smaller triangle demonstrates rotation around the origin when the origin is not a vertex.

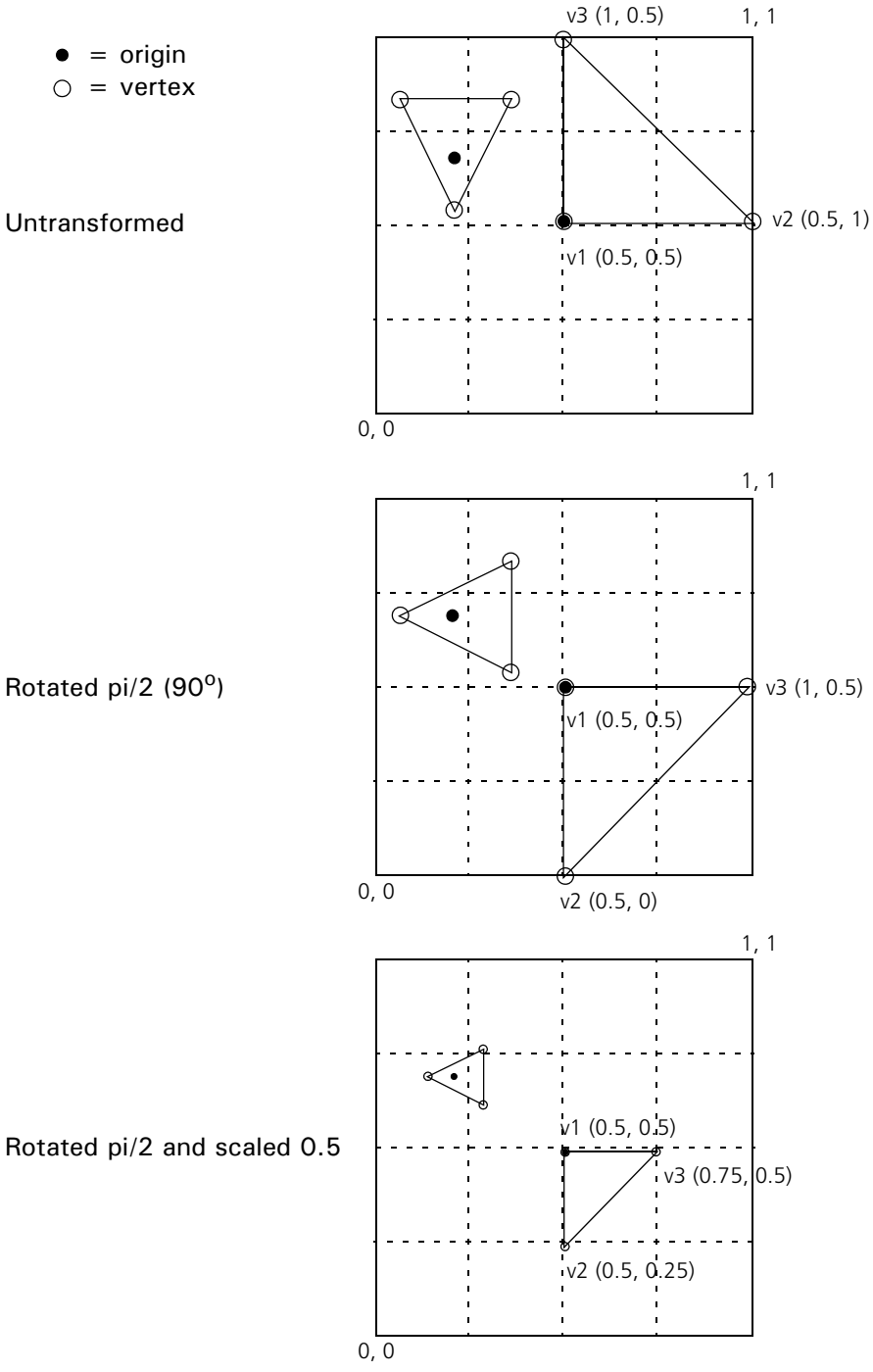


Figure 11-2. 2D triangles

11.5.5. 2D Text

DtText2DObject is used to place text on the VR-Vantage overlay. The bottom-left corner of the text is positioned using fractional screen coordinates with the *DtVdcVector2D* class. The text can have a color and can flash.

The following example creates and displays yellow text in the upper left-hand side of the 2D overlay that flashes “Hello World”.

```
DtText2DObject* text = new DtText2DObject( exConn, nextId++, “Hello
  World”, DtVdcVector2D( 0.05, 0.8 ), DtVdcColor::DtVdcYellow, true );
```

You must provide a font filename and size, in points. The default is *../data/fonts/arial.ttf* at 14 points.

11.6. 3D Objects

You can use the VR-Vantage Remote Draw API to create 3D lines, cylinders, cuboids, ellipsoids, and triangles. You can place text on the overlay based on a 3D location. All 3D positions can be specified in any one of the following ways:

- A (3D) geocentric coordinate
- Attached to a remote entity in the scene
- Attached to another draw control object.

When you attach to a line or cylinder draw control object, which itself has two positions, the position of attachment can be specified as a percentage between the two positions that make up the object being attached to. All coordinates, positions, offsets, diameters, and so on are specified in meters unless otherwise indicated. Rotations are in radians about the geocentric axes (as with normal DIS and HLA/RPR objects).

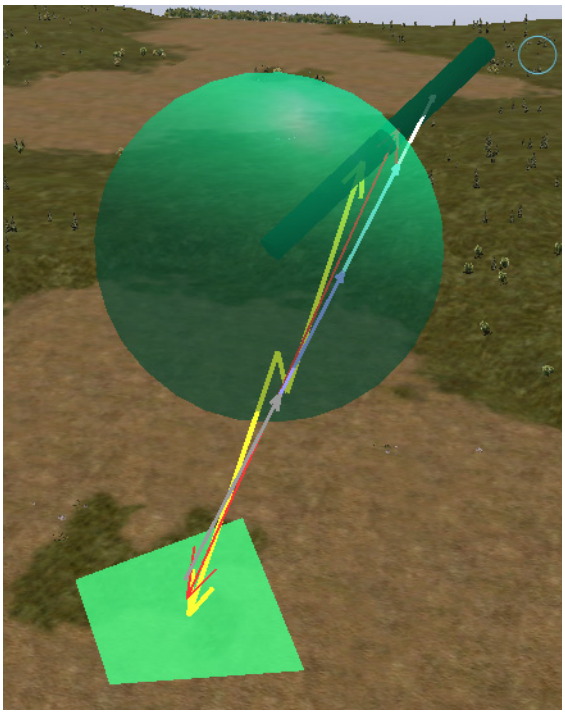


Figure 11-3. 3D objects

All 3D positions are specified using a *DtVdc3DAttachPoint*. The *DtVdc3DAttachPoint* provides a single class that can represent and manage all the different types of placement into the 3D scene.

The following example creates an attachment point in Geocentric coordinates:

```
DtVdc3DAttachPoint attach(DtVector(3139064.99, 5441231.05, 1101707.26));
```

To modify the y-coordinate using the *DtVdc3DAttachPoint*, do the following:

```
attach.setLocation(DtVector(3139064.99, 5441265.12, 1101707.26));
```

11.6.1. 3D Lines

DtVdcLine3DObject represents a line that can be positioned and displayed in the scene. The line can be divided into multiple segments of varying relative lengths. A color, line style, arrow style, pattern, and thickness can be specified for each segment. The line can flash. The line can be offset from its specified position. Optionally, the line can be terminated if it intersects any intervening terrain. In other words, the line can go from the start point to the end point or the first point at which it intersects the terrain, whichever comes first. Each endpoint of the line can be positioned using any of the member functions described for *DtVdc3DAttachPoint*.

The following example creates and displays a 3D line that goes between two entities whose IDs are in the variables globalId1 and globalId2, respectively.

```
DtVdcLine3DObject* line = new DtVdcLine3DObject( exConn, nextId++,
  DtVdc3DAttachPoint( globalId1 ), DtVdc3DAttachPoint( globalId2 ) );
```

The following example creates and displays a second 3D line that goes from the center of the line created in the previous example to a third entity, whose ID is held in the variable globalId3. It indicates that the line should terminate if it intersects the terrain.

```
DtVdcLine3DObject* line2 = new DtVdcLine3DObject( exConn, nextId++,
  DtVdc3DAttachPoint( line->id(), 50.0 ), DtVdc3DAttachPoint(globalId3));
line2->setCheckLos( true );
```

11.6.2. Cylinders

DtCylinderObject allows you to specify the attachment, shape, rotation, and color of a cylinder on the VR-Vantage screen. You can also make the object flash. You can:

- Place the cylinder at a location that becomes its center, and then specify a length and rotation
- Attach the cylinder at either end.

The diameters of the cylinder in the X and Y directions are specified in meters, using a *DtVdcVector2D*.

The following example creates and displays a cylinder that goes between two entities whose IDs are in the variables globalId1 and globalId2 respectively. It gives the cylinder a circular cross section with 0.5 meter diameter.

```
DtCylinderObject* cylinder = new DtCylinderObject( exConn, nextId++,
  DtVdc3DAttachPoint( globalId1 ), DtVdc3DAttachPoint( globalId2 ),
  DtVdcVector2D( 0.5, 0.5 ) );
```

The following example creates a second cylinder that is ten meters long. It is two meters in diameter in the X direction and one meter in diameter in the Y direction. Its center is at a point five meters above the center of the first cylinder.

```
DtCylinderObject* cylinder2 = new DtCylinderObject( exConn, nextId++,
  DtVdc3DAttachPoint( cylinder->id(), 0.5 ), 10., DtVdcVector2D( 2., 1.
  ), DtVector( 0., 0., 5. ) );
```

11.6.3. Cuboids

DtCuboidObject is used to create cuboid objects (also known as rectangular parallelepipeds). The attachment parameter specifies the center of the cuboid. A length, width, and height must be specified. The cuboid can be offset from its initial attach point and rotated. You can specify a color and flash behavior.

The following example creates a semi-transparent yellow cube with 1.5 meter sides at a specified geocentric location.

```
DtCuboidObject* cuboid = new DtCuboidObject( exConn, nextId++,
  DtVdc3DAttachPoint( DtVector( 3139064.9, 5441231.5, 1101707.3 ) ),
  1.5, 1.5, 1.5 );

DtVdcColor seeThroughYellow = DtVdcColor::DtVdcYellow;
seeThroughYellow.setAlpha( 127 );

cuboid->setColor( seeThroughYellow );
```

11.6.4. Ellipsoids

DtEllipsoidObject is used to create ellipsoids (including spheroids and spheres). The attachment parameter specifies the center of the ellipsoid. You must specify the diameters in each dimension. The ellipsoid can be offset from its initial attach point and rotated. You can specify a color and flash behavior.

The following example creates and displays a light blue ellipsoid with a length of ten meters and equal height and width of three meters. It is attached to the end of a previously defined cylinder.

```
DtEllipsoidObject* ellipsoid = new DtEllipsoidObject( exConn, nextId++,
  DtVdc3DAttachPoint( cylinder->id(), 1. ), DtVector( 10., 5., 5. ) );

ellipsoid->setColor( DtVdcColor( 0, 0, 128 ) );
```

11.6.5. 3D Triangles

DtTriangle3DObject is used to create triangles positioned in the scene. These triangles are not projected onto the 2D overlay, so 2D effects like fill patterns are not supported. You must specify the vertices of the triangle. Attachment to simulated entities or other VR-Vantage Draw Control objects is not supported. The triangle (outline) can be drawn in any color, and can flash.

The following example creates and displays a green triangle.

```
DtTriangle3DObject* triangle = new DtTriangle3DObject( exConn, nextId++,
  DtVector( 3139064., 5441231., 1101707. ),
  DtVector( 3139064., 5441231., 1101710. ),
  DtVector( 3139064., 5441236., 1101707. ),
  DtVdcColor( DtVdcColor::DtVdcGreen ) );
```

11.6.6. 3D Areas

DtArea3DObject is used to create area objects. A std::vector of *DtVector* vertices must be passed in. These vertices must be specified in geocentric coordinates, in counter-clockwise order, must be coplanar, and must specify a convex polygon. You can specify color, texture, and flashing for the area. Texture parameters include a texture directory, a texture file name, and a tiling parameter, which specifies the number of times in each dimension the texture is tiled on the area.

The following example creates the specified four-point area with a texture, tiled 10x in each direction.

```
std::vector< DtVector > fixedVerts;
DtVector pt1(-2813408.196, -4333181.164, 3728485.333);
DtVector pt2(-2813248.046, -4332934.554, 3728890.023);
DtVector pt3(-2813667.590, -4332662.117, 3728890.035);
DtVector pt4(-2813827.741, -4332908.728, 3728485.344);

fixedVerts.push_back(pt1);
fixedVerts.push_back(pt2);
fixedVerts.push_back(pt3);
fixedVerts.push_back(pt4);

DtArea3DObject* area = new DtArea3DObject( exConn, nextId++, fixedVerts
  );

area->setColor(DtVdcColor::DtVdcBlue);
area->setTextureDirectory("../data/textures/");
area->setTextureFileName( "Water_Ocean_Deep_Blue.rgb" );
area->setTextureTile( 10 );
```

11.6.7. 3D Text

Text can be positioned in the 3D scene and then projected onto the 2D overlay using *DtText3DObject*. The lower left-hand corner of the text can be positioned in 3-space using *DtVdc3DAttachPoint*. It can also be offset from that initial position in scene coordinates. The text is then projected onto the 2D overlay, so it is not occluded by terrain. You can set color and flashing attributes.

The following example creates and displays 3D text with the message “Your text here!” one meter above an existing VR-Vantage Draw Control ellipsoid. It is blue and flashing.

```
DtText3DObject* text3D = new DtText3DObject( exConn, nextId++,   “Your
  text here!”, DtVdc3DAttachPoint( ellipsoid->id() ), DtVector( 0., 0., 1
  ), DtVdcColor( DtVdcColor::DtVdcBlue ), true );
```

11.7. Attributes

The VR-Vantage Remote Draw API supports the display of groups of text:value pairs called Attributes. The values can be displayed in a variety of ways. The attributes themselves are organized into Attribute Groups. Attribute Groups can be placed in the 3D scene using ***DtVdc3DAttachPoint*** objects. Attribute Group positions are projected onto the 2D overlay, where the text and value graphics are drawn. Therefore, they are not occluded by terrain.

An Attribute Group is created by referencing an Attribute Group Template. Attribute Group Templates, in turn, are groups of Attribute Templates. Attribute Templates determine the text for each attribute that will be displayed, and the member function used to display the associated value.

Attribute Templates, Attribute Group Templates, and Attribute Groups are all VR-Vantage Draw Control objects. Attribute Group Templates are created using a constructor in the same way that other VR-Vantage Draw Control objects are created. Attribute Templates are then created using member functions provided by the Attribute Group Templates.

VR-Vantage must know about an Attribute Group Template and its corresponding Attribute Templates before an Attribute Group can be created that refers to that Attribute Group Template.

Attribute Groups are then created by referencing an existing Attribute Group Template, and positioned.

Figure 11-4 illustrates a view of the Attribute Group Templates tab of the Remote Graphics Panel in VR-Vantage and the display of attributes for the entity and packet attributes groups.

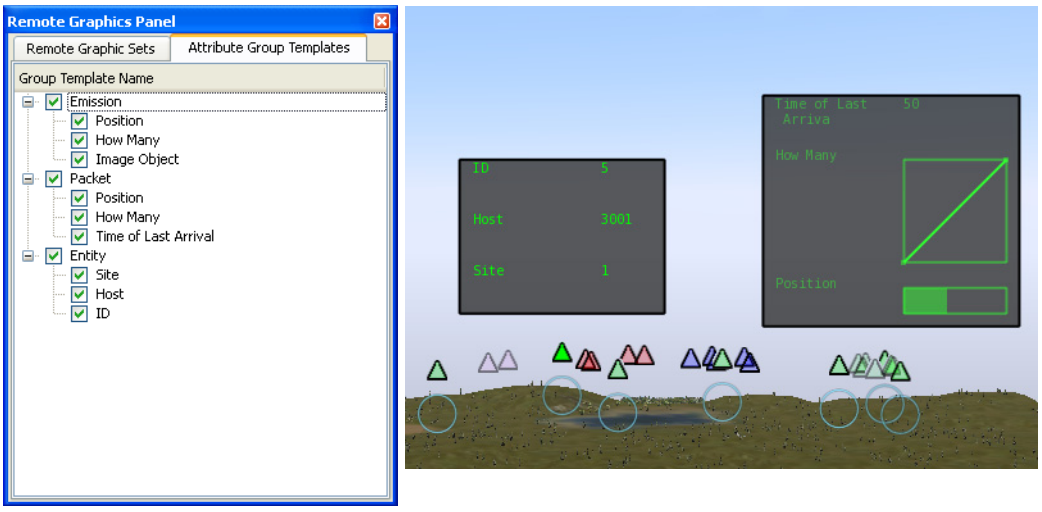


Figure 11-4. Attribute group template display

11.7.1. Attribute Group Templates

The *DtAttributeGroupTemplateObject* class represents Attribute Group Templates. The constructor takes an ID (like all VR-Vantage Draw Control Objects), and a name.

11.7.2. Attribute Templates

To create an Attribute Template, use the *DtAttributeGroupTemplateObject* member function createAttributeTemplate(). It has two forms. Use the first form to create all Attribute Template types except for the Bar type Attribute Template, which requires two extra parameters. Attribute Template types are represented by the DtAttributeTemplateObject::DtVdcAttributeType enumeration. Table 11-1 describes the DtAttributeTemplateObject templates.

Table 11-1: DtAttributeTemplateObject templates

Attribute Template Type	Description
DtVdcAttributeTypeNoValue	The NoValue Attribute Template type allows for attributes with no associated value (just the Attribute Template name is displayed.)
DtVdcAttributeValueNumeric	The ValueNumeric Attribute Template type displays a value in its numeric form.
DtVdcAttributeTypeValueBar	The ValueBar Attribute Template type displays a value using a bar that indicates where the value falls relative to a specified minimum and maximum value.
DtVdcAttributeTypeValueGraph	The ValueGraph Attribute Template type maintains the last 10 X,Y values sent and displays them as a line graph.
DtVdcAttributeTypeBitmap	The Bitmap Attribute Template type displays a bitmap (.bmp file) whose name is the attribute value (if given as a relative path, the filename is relative to the directory VR-Vantage was run from.)

The following example creates and tells VR-Vantage about an Attribute Group Template named “Entity,” which contains two Attribute Templates: a ValueNumeric named “Speed:” and a ValueBar named “Health:” that goes from a minimum value of 0.0 to a maximum value of 1.0.

```
// First, create the Attribute Group Template
DtAttributeGroupTemplateObject* attGrpTemplate = new
  DtAttributeGroupTemplateObject( exConn, nextId++, “Entity” );

// Next, create each Attribute Template
attGrpTemplate->createAttributeTemplate( “Speed: “,
   DtAttributeTemplateObject::DtVdcAttributeValueNumeric );

// The interface knows that this is a ValueBar attribute because
// it specifies min/max values
attGrpTemplate->createAttributeTemplate( “Health: “,
   0., 1. );
atGrpTemplate->update();
```

11.7.3. Attribute Groups

DtAttributeGroupObject is used to represent an Attribute Group. An Attribute Group is an instance of an Attribute Group Template. It is positioned in 3-space using a *DtVdc3DAttachPoint*, and can be offset from that. A pointer to the referenced Attribute Group Template is provided to the Attribute Group's constructor. You can specify the color used to display Attribute Group's name and attributes and can set it to flash.

Attribute instances are only created by VR-Vantage. Values for specific attributes are set through the Attribute Group object by referring to the ID of the Attribute Template that corresponds to the Attribute value you want to set.

The following example creates an Attribute Group attached to a *DtReflectedEntity* pointed to by pEntity, based on the Attribute Group Template created in "Attribute Templates," on page 11-19. The example sets the Speed Attribute value to the speed of the reflected entity it is attached to, and sets the Health Attribute to one. It displays the Attribute Group.

```
DtAttributeGroupObject* attGroup = new
  DtAttributeGroupObject( exConn, nextId++, attGrpTemplate,
  DtVdc3DAttachPoint( entity->globalId() ) );

attGroup->setNumericAttributeValue( "Speed: ",
   sqrt( entity->esr()->velocity().magnitudeSquared() ) );

attGroup->setValueBarAttributeValue( "Health: ", 1. );

c++Group->update();
```

To update the speed Attribute for the Attribute Group and inform VR-Vantage, do the following:

```
attGroup->setNumericAttributeValue( "Speed: ",
   sqrt( entity->esr()->velocity().magnitudeSquared() ) );

atGroup->update();
```

To hide all speed attributes created by using the Attribute Group Template, do this:

```
DtVrvControlMessageMarshaller<DtDataInteraction> marshaller;
DtRemoteGraphicSet_v1 rgCommand;
DtString name = attGroupTemplate.name();
name += "::Speed: ";
sprintf(rgCommand.mySetName, "%s", name.c_str());
rgCommand.myState = true;
rgCommand.myAppliesToAttribute = true;

DtDataInteraction dataInter;
marshaller.encode(&rgCommand, &dataInter);
exConn.sendStamped(dataInter);
```

11.8. Remote Graphic Sets

You can group VR-Vantage Draw Control objects to control their visibility. Showing a set is equivalent to showing each object and any other subordinate sets. These sets are communicated to VR-Vantage and appear in VR-Vantage's Remote Graphics panel. This panel allows sets to be hidden and shown.

Draw control objects can be added to or removed from sets. You can also add subordinate sets to a set. A set is created implicitly when you add an object to it.

The following example adds two draw control objects to the Squares set. They are pointed to by DtDrawControlObject* pointers square1 and square2.

```
square1->setRemoteGraphicSet(“Squares”);
square2->setRemoteGraphicSet(“Squares”);
```

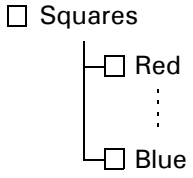
To hide everything on the shapes layer, do this:

```
DtVrvControlMessageMarshaller<DtDataInteraction> marshaller;
DtRemoteGraphicSet_v1 rgCommand;
sprintf(rgCommand.mySetName, "Squares");
rgCommand.myState = false;

DtDataInteraction dataInter;
marshaller.encode(&rgCommand, &dataInter);
exConn.sendStamped(dataInter);
```

To show the (now hidden) squares using the squares layer:

```
DtVrvControlMessageMarshaller<DtDataInteraction> marshaller;
DtRemoteGraphicSet_v1 rgCommand;
sprintf(rgCommand.mySetName, "Squares");
rgCommand.myState = true;

DtDataInteraction dataInter;
marshaller.encode(&rgCommand, &dataInter);
exConn.sendStamped(dataInter);
```

You can group sets hierarchically using double colon (::) syntax. For example, “Squares::Blue” and “Squares::Red” would result in the following hierarchy:



Menu Names

This appendix lists the menu names that you need to know to customize menus.

A.1. Menu Names

Using the VR-Vantage Toolkit, you can add items to the menus of a VR-Vantage application. However, to do this, you must know the name of the menu option next to which you want to add the new one. This section lists all of the menu commands in the VR-Vantage applications provided by VT MÄK.

Some menu commands apply only to certain versions of VR-Vantage, such as VR-Vantage FreeView.

File Menu

- DtNewSceneItem
- DtOpenSceneItem
- DtSaveSceneItem
- DtSaveSceneAsItem
- DtNewTerrainItem
- DtOpenTerrainItem
- DtSaveTerrainItem
- DtSaveTerrainAsItem
- DtCloseTerrainItem
- DtAddTerrainPatchMenuItem
- DtAddFeatureLayerItem
- DtNewStealthWindowItem
- DtNewPlanViewWindowItem
- DtExitItem.

View Menu

- DtObjectInfoItem
- DtFullScreenItem.

Settings Menu

- DtDisplaySettingsMenuItem
- DtApplicationSettingsMenuItem
- DtSceneSettingsMenuItem
- DtTerrainSettingsMenuItem
- DtConnectionsMenuItem
- DtEntityTypeMappingsMenuItem
- DtCigiMappingsMenuItem
- DtVisualEditorsMenuItem
- DtEntityLabelsItem
- DtTacticalGraphicsItem
- DtTacticalGraphicsLabelsItem
- DtEntityHatLinesItem
- DtShowAggregatesItem
- DtFireDetonateLinesItem
- DtGroundClampingItem
- DtSensorVolumeItem
- DtCockpitSettingItem
- DtWireframeItem
- DtShadowsItem
- DtTerrainScalingItem
- DtRemoteGraphicsItem.

Observer Menu

- DtObserverModeSettingItem
- DtObserverModelSetItem
- DtSpeedScalingItem
- DtTerrainDraggingItem
- DtObserverViewControlItem
- DtDisplayCompassItem
- DtDisplayGridLinesItem
- DtConstrainToTerrainItem
- DtForceNorthOrientationItem
- DtModelScalingItem
- DtSkyItem
- DtDisplayFeaturesItem
- DtDustCloudItem
- DtWakeItem
- DtContrailItem
- DtFootprintItem
- DtTreadItem
- DtFireItem
- DtSmokePlumeItem
- DtDetonationItem
- DtMuzzleFlashItem
- DtTrackHistoryItem
- DtObserverAdvancedItem.

Intervisibility Menu

- DtIntervisibilityItem
- DtIntervisibilityLineMenuItem
- DtIntervisibilityFanMenuItem
- DtIntervisibilityEntityMenuItem
- DtIntervisibilityDeleteMenuItem.

Context-Sensitive Menus

- DtAttachMimicItem
- DtAttachTrackItem
- DtAttachFollowItem
- DtAttachMimicTrackItem
- DtAttachTetherTrackItem
- DtAttachTetherItem
- DtCollapseAggregateMenuItem
- DtCollapseAllAggregateMenuItem
- DtExpandAllAggregateMenuItem
- DtExpandAggregateMenuItem
- DtCreateInsetViewItem.

Model Menu (VR-Vantage FreeView)

- DtOpenModelItem
- DtCloseModelItem.

Help Menu

- DtHelpItem
- DtAboutItem
- DtProductsWebSiteMenuItem
- DtVersionWebSiteMenuItem
- DtFreeViewHelpItem.

Other Menus

- DtViewSettingsItem
- DtOsgEarthServerItem
- DtDeselectAllMenuItem
- DtSeparatorItem
- DtCreateObjectMenuItem
- DtShowHideItem
- DtObserverMenuModelSetItem
- DtLogItem
- DtTransparencyToggleItem
- DtExampleSettingsMenuItem
- DtOsgEarthStartupItem
- DtDisConnectionsItem

- DtCloseSceneItem
- DtDeleteRangeLineMenuItem
- DtTacticalGraphicsHatLinesItem
- DtCreateRangeLineMenuItem
- DtNewWindowItem
- DtMainModelItem
- DtObserverModeItem
- DtCigiConnectionsItem
- DtExampleToolbarMenuItem.

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Link - Simulate - Visualize

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