Scalable VR Application Authoring: IEEE VR 2003 Tutorial Course Notes
by The VR Juggler Team
Published March 23, 2003
# Table of Contents

## I. Getting Started

1. Installing VR Juggler ................................................................. 1
   1. Installing from a Compressed TAR File .......................... 2
   2. Installing from a ZIP File (Win32 only) ....................... 3
2. Environment Variables ............................................................ 4
   1. How to Set Environment Variables ................................. 4
      1.1. Common Conventions and Background .................. 4
      1.2. C-Style Shells (csh, tcsh) ................................. 4
      1.3. sh-Derived Shells (sh, ksh, bash, zsh, etc.) ........... 5
      1.4. DOS Shell ..................................................... 5
      1.5. Win 32 GUI .................................................. 6
      1.6. Syntax Used in this Document .............................. 9
   2. Required Environment Variables ...................................... 10
   3. Optional Related Environment Variables ......................... 11
3. VR Juggler Sample Applications ............................................. 12
   1. Tutorial Applications ...................................................... 12
   2. Advanced OpenGL Performer Applications .................... 12
4. Compiling a VR Juggler Sample Program ................................. 13
   1. Required Reading .......................................................... 13
   2. Compiling an Application .............................................. 13
      2.1. Compiling from the Command Line ....................... 13
      2.2. Compiling Using Microsoft Visual Studio ............... 13
5. Running a VR Juggler Sample Program ...................................... 19
   1. Required Reading .......................................................... 19
   2. Running an Application on the Desktop .......................... 19
   3. Running an Application in a VR System .......................... 24

## II. Application Programming

6. Application Basics ................................................................. 26
   1. Application Object Overview ........................................ 26
      1.1. No main()—Don't call me, I'll call you ................ 26
      1.2. Application Objects Derive from Base Classes for Specific Graphics APIs 26
      1.3. Writing an Application Means Filling in the Blanks .... 27
   2. Benefits of Application Objects ........................................ 28
      2.1. Allow for Run-Time Changes ................................. 28
      2.2. Low Coupling .................................................. 28
      2.3. Allows Implementation Changes ............................ 28
      2.4. Multi-Language Interaction ................................... 28
   3. VR Juggler Startup .............................................................. 29
      3.1. No main()—Sort Of .......................................... 29
      3.2. Structure of a main() Function ............................ 29
   4. Kernel Loop ................................................................. 30
      4.1. Definition of a Frame .......................................... 31
   5. Base Application Object Interface ....................................... 32
      5.1. Initialization .................................................... 33
      5.2. Frame Functions ............................................... 34
   6. Draw Manager-Specific Application Classes ......................... 35
      6.1. OpenGL Application Class .................................. 35
      6.2. OpenGL Performer Application Class ................... 37
   7. Helper Classes .................................................................. 40
      1. The gmtl::Vec<S, T> Helper Class ............................ 40
         1.1. High-Level Description .................................... 40
         1.2. Using gmtl::Vec3f and gmtl::Vec4f .................... 41
         1.3. Creating Vectors and Setting Their Values .......... 41
List of Figures

2.1. Windows 2000 System Properties Dialog ................................................................. 6
2.2. Windows 2000 Environment Variable Editor Dialog .............................................. 7
2.3. Setting VJ_BASE_DIR on Windows 2000 ................................................................. 8
4.1. Selecting the Visual C++ Project File .................................................................. 14
4.2. MPApp Project ...................................................................................................... 14
4.3. Project Menu ......................................................................................................... 15
4.4. Project Settings Dialog ......................................................................................... 16
4.5. Build MPApp.exe .................................................................................................. 17
4.6. Execute MPApp.exe .............................................................................................. 18
5.1. MPApp running on a Linux desktop with multiple input windows ................. 21
5.2. MPApp running on a Linux desktop with one window ......................................... 21
6.1. vrj::App hierarchy ............................................................................................... 26
6.2. Kernel loop sequence .......................................................................................... 30
6.3. Application object interface .................................................................................. 32
6.4. vrj::GlApp interface extensions to vrj::App ......................................................... 35
6.5. vrj::PfApp interface extensions to vrj::App .......................................................... 37
8.1. VR Juggler kernel control loop ............................................................................ 60
8.2. vrj::GlApp application class .............................................................................. 64
8.3. VR Juggler OpenGL system ............................................................................... 71
8.4. vrj::PfApp application class ................................................................................ 73
8.5. vrj::OpenSGApp application class ....................................................................... 79
Part I. Getting Started

To begin, we explain how to get started with VR Juggler in general. We first describe how to install VR Juggler after it has been downloaded (see http://www.vrjuggler.org/download.php to get the latest version). We provide installation instructions for both UNIX-based and Windows-based operating systems. We also explain the use of environment variables including how to set them and which ones need to be set on a given platform.

The most important aspect of this introductory part is the discussion on VR Juggler application execution. We explain first how to compile the sample applications that come with the VR Juggler distributions. We then review the desktop input mechanism and explain how to use it with any VR Juggler application.
Chapter 1. Installing VR Juggler

As with most Open Source projects, VR Juggler is distributed as compressed archive files using popular formats. Installing a distribution requires very little effort, but you do need to know how to use archiving utilities to extract the installation tree. Automation of the installation is a goal of the VR Juggler team, but we are still finalizing the details of cross-platform installation management. Before reading further, you should know where you want to install VR Juggler, and you should make sure that you have access to write to that directory.

1. Installing from a Compressed TAR File

The TAR (Tape ARchive) format has been around for a long, long time in the UNIX world. It is simply a collection of files in a directory tree that are lumped into a single file suitable for writing to a tape or for downloading. The format is a standard, and the tar utility is available on every UNIX-based platform and on Win32. A free version can be downloaded from the GNU Project [http://www.gnu.org/]. A compressed TAR file is made for each VR Juggler distribution, and some distributions come in other formats as well. You can always count on the availability of a TAR file, though. The TAR files are compressed using either GZIP or BZIP2, both of which are standard compression formats. The gzip utility is freely available from the GNU Project, and the bzip2 utility can be down loaded for free from RedHat, Inc. [http://sources.redhat.com/bzip2/]. The GNU version of TAR has the GZIP and BZIP2 algorithms built in. The compression algorithm used can be determined by the file extension. Files compressed with GZIP end in .gz; files compressed with BZIP2 end in .bz2.

Once you have downloaded a VR Juggler TAR distribution, you can unpack it one of two ways depending on what your platform's version of TAR supports. Before extracting the installation tree, make sure that your current directory is the one where you want to install VR Juggler. If your version of TAR does not have GZIP built in (it does not support the -z option), the following command will do the decompression and extraction:

```
% gzip -cd vrjuggler-distribution.tar.gz | tar -xvf -
```

For versions of TAR without built-in BZIP2 support (there is no -j option) the command is similar:

```
% bzip2 -cd vrjuggler-distribution.tar.bz2 | tar -xvf -
```

Here, you should fill in `vrjuggler-distribution.tar.gz` (or `vrjuggler-distribution.tar.bz2`) with the name of the VR Juggler distribution file you downloaded. The above commands will work with any shell that supports redirection of standard output to a pipe. If that looks too scary, you can separate the decompression and extraction into two commands (for GZIP):

```
% gunzip vrjuggler-distribution.tar.gz
% tar -xvf vrjuggler-distribution.tar
```

or for BZIP2:

```
% bunzip2 vrjuggler-distribution.tar.bz2
% tar -xvf vrjuggler-distribution.tar
```

Note that the distribution file in the second command does not have the .gz extension after `gzip` (1) is run. These steps also work if your version of `tar` supports the -z option (-j for BZIP2), but you can simplify your work if that option is supported. The following illustrates how to decompress and extract a TAR file compressed with GZIP all in one step:

```
% tar -xzvf vrjuggler-distribution.tar.gz
```
The following would be used for a TAR file compressed with BZIP2:

\% tar -xjvf vrjuggler-distribution.tar.bz2

In either case, while the command runs, you will see the name of each file as it is written to disk. This is because of the \-v option to `tar\(1\)` that tells it to be verbose in its efforts, `tar\(1\)` takes care of creating all the directories in the installation tree, so you only need to have the base directory (for example, `/usr/local`) when you start. For more information about these utilities, please refer to the `tar\(1\)` and `gzip\(1\)` manual pages.

2. Installing from a ZIP File (Win32 only)

On the Win32 family of platforms, the ZIP format rules. In the old days, you would use the PKZIP utility to uncompress and extract a ZIP file. Nowadays, most people use WinZip [http://www.winzip.com/] or some other comparable graphical interface. This documentation covers only the use of WinZip when extracting a ZIP file.

Once you have downloaded the VR Juggler ZIP file, the easiest way to extract it is to double-click on its icon in the open folder window. Double-clicking opens the main WinZip window. Click the Extract button to open the extraction dialog box. In this dialog, choose the directory where VR Juggler will be installed and click Extract. WinZip will then proceed to extract the ZIP file into the directory you named.
Chapter 2. Environment Variables

There are several environment variables that affect the way VR Juggler works. Some of these are required to compile and run applications while others are optional. This chapter lists all such variables and explains their meanings and uses.

1. How to Set Environment Variables

The syntax for setting or changing an environment variable varies with operating systems and shell interpreters. Instead of choosing one style of syntax that is specific to a particular shell type, we define our own syntax which you must then translate to your shell's specific syntax. Before defining this syntax, we present the method used to set environment variables in the three most common types of shells. We also provide a quick overview of how to set environment variables using Win32-based GUIs.

1.1. Common Conventions and Background

A convention used throughout this book is to name the variables using all capital letters. In almost all cases, regard less of the shell, this is the naming convention used for environment variables.

Setting a path with an environment variable can require special syntax. Because of this, the method for doing so may vary from shell to shell. Paths are important with VR Juggler when looking up the path to a shared library (dynamically linked library). For each shell, the syntax for setting a path is given.

Referring to environment variables can also vary from shell to shell. An example of how to print the value of an environment variable will be given for each shell. An example of how to refer to an environment variable is also provided as these two operations may vary even within one kind of shell!

In all shells, an environment variable is only available within that single shell instance. That is, setting an environment variable at a command prompt only affects that specific shell and will not be available from other concurrent or future shells. To make a setting “permanent”, it should be done in file read by all shell instances when they are started. This is addressed briefly as appropriate for each shell type.

1.2. C-Style Shells (csh, tcsh)

In a C-style shell (i.e., one whose interface is based on the C programming language), setting environment variables is done using the built-in command `setenv`. It is used as follows:

```bash
% setenv <VARIABLE_NAME> <value>
```

where the string `<VARIABLE_NAME>` represents the name of the variable you are going to set and `<value>` represents the value assigned to that variable. Both are required. If the named variable did not exist before, it will pop into existence. Otherwise, you overwrite the old setting with the new one.

To print the value of an environment variable, use the following command:

```bash
% printenv <VARIABLE_NAME>
```

Referring to a variable, however, is done using the following syntax:

```bash
% cd $VARIABLE_NAME/bin
```

Paths are specified as a colon-separated list. An example of this is:
For these types of shells, a “permanent” setting for a given variable should usually be done in your .cshrc file or in your .login file, both of which should be in your home directory. In most cases, it is better to use .cshrc because it is evaluated for every shell instance.

1.3. sh-Derived Shells (sh, ksh, bash, zsh, etc.)

In a shell based on sh, setting environment variables is done using the built-in command export. It is used as follows:

% export <VARIABLE_NAME>=<value>

or

% <VARIABLE_NAME>=<value>
% export <VARIABLE_NAME>

Here, the string <VARIABLE_NAME> represents the name of the variable you are going to set and <value> represents the value assigned to that variable. Both are required. Note that there is no space between the variable name and its value. If the named variable did not exist before, it will pop into existence. Otherwise, you overwrite the old setting with the new one. If the variable was already among your current shell’s environment variables, the export command is not necessary.

To print the value of an environment variable, use the following command:

% echo $VARIABLE_NAME

Getting the value of a variable works the same way.

Paths are specified as a colon-separated list. An example of this is:

% echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin

For these types of shells, a “permanent” setting for a given variable should usually be done in the .profile file in your home directory or in your shell’s “rc” file. Different shells have different names for this file. Examples are .bashrc for BASH and .zshrc for Zsh. Please refer to your shell’s documentation for more information. In any case, the file will be in your home directory.

1.4. DOS Shell

The typical syntax for setting an environment variable from the command line (in a DOS shell window) under Win32 is:

C:\> set <VARIABLE_NAME>=<value>

Here, <VARIABLE_NAME> is the name of the environment variable to be set, and <value> is the value being assigned to that variable. If the named variable did not exist before, it will pop into existence. Otherwise, you overwrite the old setting with the new one.

To print the value of an environment variable, use the following command:
C:\> set <VARIABLE_NAME>

Referring to a variable, however, is done using the following syntax:

C:\> cd %VARIABLE_NAME%\bin

Paths are specified as a semicolon-separated list. An example of this is:

C:\> set PATH
C:\WINDOWS;C:\bin;C:\

For some versions of Windows, a “permanent” setting for a given variable should usually be done in C:\AUTOEXEC.BAT. In newer versions (Windows ME in particular) and in the Windows NT line of operating systems, the setting is done using the Control Panel. Please refer to the next section for more information on that method.

1.5. Win 32 GUI

Before reading this section, please be sure to have read Section 1.4, “DOS Shell”. This is necessary because the Win32 GUI for setting environment variables is simply a front-end to that older method and thus uses the same conventions and syntax. The versions of Windows to which this subsection applies are indicated individually since each is a little different. For more detailed information, please refer to the Windows online help system and search for “environment variables”.

1.5.1. Windows 2000

In the Control Panel, open the System icon. Under the Advanced tab, there is a button labeled Environment Variables (shown in Figure 2.1). Clicking this button opens the dialog box shown in Figure 2.2. Here, you can set variables for yourself and, if you have the access privileges, for all users.

Figure 2.1. Windows 2000 System Properties Dialog
Figure 2.2. Windows 2000 Environment Variable Editor Dialog
To set up VR Juggler, the environment variable VJ_BASE_DIR needs to be set, probably for all users, though it depends very much on local system requirements. For this example, let us say that the Win32 version of VR Juggler is installed in the D:\ directory. As such, we will set VJ_BASE_DIR as shown in Figure 2.3. Note that we use a forward slash (/) instead of a backslash (\) for the directory separator. This is not required, though it is helpful in reducing confusion with respect to other examples.

Figure 2.3. Setting VJ_BASE_DIR on Windows 2000
1.5.2. Windows NT 4.0

In the Control Panel, open the System icon. The window that is opened has a tab labeled Environment Variables. Here, you can set variables for yourself and, if you have the access privileges, for all users. The GUI is similar to that shown above for Windows 2000.

1.6. Syntax Used in this Document

To avoid tying this documentation to a single style of environment variable creation, assignment and reference, the following syntax will be used exclusively from this point onward. Please read this carefully before proceeding.

1.6.1. Naming Environment Variables

When naming an environment variable in the plain text of this document, the variable will be referred to by its name only. For example, to talk about the environment variable containing your path, we will talk about it as PATH.

1.6.2. Creating/Setting Environment Variables

The syntax to set an environment variable is:

% <VARIABLE_NAME> = <value>

Setting an environment variable also creates it if it is not already present in the current shell's environment.

1.6.3. Printing an the Value of an Environment Variable

Printing an environment variable's value to standard output (stdout) is done as follows:

% echo $VARIABLE_NAME

value

1.6.4. Referring to the Value of an Environment Variable

To get the value of an environment variable when it needs to be expanded, the following syntax will be used:

% cd $VARIABLE_NAME/bin

Here, the reference to the value is $VARIABLE_NAME.
2. Required Environment Variables

VJ_BASE_DIR
The environment variable VJ_BASE_DIR tells a VR Juggler application where to find important data files. It is required to compile and run any Juggler application. It should be set to the base directory of the installed VR Juggler library. For example, if you downloaded a UNIX version of VR Juggler 2.0 and extracted it to the directory `/home/software/`, you would set VJ_BASE_DIR with this command:

```
% VJ_BASE_DIR = /home/software/vrjuggler-2.0
```

The last component of the path depends on the particular version of Juggler you have downloaded.

If you downloaded and built VR Juggler from the source code, the compilation creates a directory called `instlinks` which can be used as a VR Juggler base:

```
% VJ_BASE_DIR = $HOME/juggler/my_build_dir/instlinks
```

In any case, on a Win32 platform, you should use `/`s as the path separator for VJ_BASE_DIR rather than `\`s. The compiler tools can handle either, and the utilities in juggler-tools will behave much better if UNIX-style paths are used. It is safe to use the drive letter at the start of the path (e.g., `C:/software/vrjuggler-2.0`).

PATH
To compile any of the sample applications, the `$VJ_BASE_DIR/bin` directory must be added to your PATH as follows:

```
% PATH = $PATH:$VJ_BASE_DIR/bin
```

Depending on your shell, you may need to run the `rehash` command after executing the above.

Windows users must also include the `$VJ_BASE_DIR/lib` directory in their PATH setting. This is so that the VR Juggler DLLs will be found when an application is executed.

JDK_HOME
The JDK_HOME environment variable is required by the script that starts VRJConfig, the VR Juggler configuration program. If Java is installed on your system, JDK_HOME may already be set. If not, it needs to be set to the base of the Java installation.

LD_LIBRARY_PATH (UNIX/Linux only), LD_LIBRARYN32_PATH (IRIX only), LD_LIBRARY64_PATH (IRIX only)
UNIX/Linux systems use these environment variables to find dynamically loaded libraries, such as `libJuggler.so`. Unless you are building everything with static libraries, you will need to set these to include the VR Juggler library directory (under VJ_BASE_DIR). IRIX supports several Application Binary Interfaces (ABIs). VR Juggler supports only the N32 and 64 formats, and there are different library path variables for each. The N32 ABI uses the LD_LIBRARYN32_PATH variable, and the 64 ABI uses LD_LIBRARY64_PATH. An example of setting the library path is as follows:

```
% LD_LIBRARY_PATH = $VJ_BASE_DIR/lib
```

Note
On some SGI systems running IRIX, users of the MIPSpro Compilers (version 7.3) will need to add another directory as follows:

```
% LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/usr/lib32/cmplrs:$VJ_BASE_DIR
```

### 3. Optional Related Environment Variables

**VPR_DEBUG_NFY_LEVEL**

This variable can be used to control the amount of diagnostic information a VR Juggler application outputs. Its value is a number between 0 (only very important messages are printed) and 7 (vast amounts of data) inclusive. Non-hackers are advised to use levels 0 through 3, as higher debug levels become increasingly cryptic and can severely impact application performance. The default is level 1—only errors and critical information are output. An example of setting a value for this variable is:

```
% VPR_DEBUG_NFY_LEVEL = 3
```

**VPR_DEBUG_ALLOW_CATEGORIES**

This variable can be used to control which components of VR Juggler are allowed to output diagnostic data. If for some reason you set `VPR_DEBUG_NFY_LEVEL` to 5 or higher, this variable can be used to filter the output. The value of `VPR_DEBUG_CATEGORIES` is a space-separated list of Juggler debug component names (defined in `$VJ_BASE_DIR/include/vrj/Util/Debug.h`, `$VJ_BASE_DIR/include/vpr/Util/Debug.h`, `$VJ_BASE_DIR/include/tweek/Util/Debug.h`, `$VJ_BASE_DIR/include/jccl/Util/Debug.h`, and `$VJ_BASE_DIR/include/gadget/Util/Debug.h`). The default value is “DBG_ALL”, which performs no filtering whatsoever. Examples of setting it are as follows:

```
% VPR_DEBUG_ALLOW_CATEGORIES = DBG_ERROR
% VPR_DEBUG_ALLOW_CATEGORIES = "DBG_KERNEL DBG_INPUT_MGR DBG_DRAW_MGR"
% VPR_DEBUG_ALLOW_CATEGORIES = "DBG_CONFIG DBG_RECONFIGURATION"
```

**VPR_DEBUG_DISALLOW_CATEGORIES**

This variable is basically the opposite of `VPR_DEBUG_ALLOW_CATEGORIES`. Instead of specifying which debugging categories you want to see, you specify which ones you do not want to see. Its default value is empty which means that no debugging categories are excluded. Examples of setting it are as follows:

```
% VPR_DEBUG_DISALLOW_CATEGORIES = DBG_ERROR
% VPR_DEBUG_DISALLOW_CATEGORIES = "DBG_KERNEL DBG_INPUT_MGR DBG_DRAW_MGR"
% VPR_DEBUG_DISALLOW_CATEGORIES = "DBG_CONFIG DBG_RECONFIGURATION"
```
Chapter 3. VR Juggler Sample Applications

VR Juggler comes with several sample applications in its samples directory tree. Many of them are very simple and are designed to demonstrate a specific feature of VR Juggler or a technique to use when writing your own applications. This chapter lists the current sample applications as of this writing and gives a quick description of what you as a potential developer might find interesting in the code. Those users who just want to run applications can safely skip this chapter.

1. Tutorial Applications

Some sample applications designed for getting started with VR Juggler are found in $VJ_BASE_DIR/share/vrjuggler/samples/OGL/simple. All of these applications were designed to be used as part of courses teaching people how to write VR Juggler applications using OpenGL. They contain clear comments explaining what the code is doing, and they are intended to be as simple as possible. These tutorials are as follows:

- simpleInput: An application that demonstrates how to get input from devices. No graphics are rendered with this application. It is intended to be a starting point for getting an understanding of how user input is queried.

- SimpleApp: A very simple OpenGL application that draws a small cube in space and draws the coordinate axes for the cube.

- contextApp: An application demonstrating how to use OpenGL display lists in VR Juggler applications. This extends SimpleApp by using a display list to draw a cube and by moving the cube with the wand.

- ConfigApp: A relatively simple application that demonstrates how user-level code can take advantage of the VR Juggler configuration system, JCCL.

- MPApp: A more complex OpenGL application that demonstrates how to do multi-processing in VR Juggler applications. As it exists in its distributed form, no multi-processing is done. A more detailed lesson is available that explains how to extend the application to employ multi-processing techniques.

Part II of this book explains how to use the above to learn VR Juggler application programming. It contains sections explaining each of the above applications in great detail. Each lesson ends with an exercise where the reader extends the application to include some new functionality.

2. Advanced OpenGL Performer Applications

Examples of OpenGL Performer applications can be found in $VJ_BASE_DIR/share/vrjuggler/samples/Pf/advanced. These are for more advanced developers who are familiar with Performer and some of the more complicated aspects of VR Juggler. There are two main programs to be found there:

- pfNav: A starting point for basic VR Juggler Performer applications that need to load a model and navigate through it. Users implement their application by inheriting from a provided class, simplePfNav. This may be a good place for intermediate-level users of OpenGL Performer to start because simplePfNav hides many of the complicated details (which actually makes that class far from simple).

- pfConfigNav: A more advanced example of a VR Juggler Performer application that can be given its model through a VR Juggler configuration element.
Chapter 4. Compiling a VR Juggler Sample Program

Now that you have VR Juggler installed and you have your environment all configured, it is time for the fun to begin. No, seriously. You are now ready to compile and run VR Juggler applications, and that is the whole point, right? This chapter explains how to compile the applications provided in the directory $VJ_BASE_DIR/share/vrjuggler/samples/OGL/simple.

1. Required Reading

Before reading any further, make sure you have already read the instructions on how to install VR Juggler (in Chapter 1, Installing VR Juggler) and on how to configure your environment (in Chapter 2, Environment Variables). That information will not be repeated, and it is assumed that you already know what we mean by $VJ_BASE_DIR. You should also have a basic understanding of how make(1) works, but in these examples, nothing more will be necessary than typing make on the command line. Refer to the make(1) manual page for more information about it.

2. Compiling an Application

There are two ways to compile VR Juggler applications: from the command line or with Microsoft Visual Studio. Compiling an application on the command line requires the use of GNU make (often installed as gmake) so that it will work on all supported platforms including Win32. Using Microsoft Visual Studio will only work on Win32.

2.1. Compiling from the Command Line

All the sample programs in $VJ_BASE_DIR/share/vrjuggler/samples use the same basic steps to compile unless otherwise noted. Always refer to the top of the sample application's Makefile for information that may be specific to building that application. In general, though, all applications' makefiles require the GNU version of the make(1) utility, sometimes installed as gmake.

The example used here will be the MPApp tutorial application found in $VJ_BASE_DIR/share/vrjuggler/samples/OGL/simple/MPApp. It is an OpenGL-based application that will compile and run on all platforms supported by VR Juggler. Begin by changing into the directory $VJ_BASE_DIR/share/vrjuggler/samples/OGL/simple/MPApp in a command shell.

To compile MPApp, simply enter the following:

% gmake

The compile process will then begin. As noted above, the use of GNU make is required to use the distributed make# files. With Cygwin, GNU make is simply make. If you have your system set up properly, it will complete with an executable MPApp file (or MPApp.exe on Win32) in the directory. Now that you have a program compiled, it is time to learn how to run it. (Readers who are not using Visual Studio can skip ahead to Chapter 5, Running a VR Juggler Sample Program.)

2.2. Compiling Using Microsoft Visual Studio

Note

Remember that the Netscape Portable Runtime (NSPR) is required to use VR Juggler on Windows. Its DLL directory must be in your path (via the PATH environment variable) for proper application execution. The NSPR can be downloaded from the NSPR home page [http://www.mozilla.org/projects/nspr].
All OpenGL sample applications are shipped with pre-configured Microsoft Visual C++ projects. This is done to help new users get started with compiling VR Juggler applications and to give experienced Visual Studio users a starting place for their application development. To use the workspace for the MPApp application, begin by opening the folder containing the source code and double-clicking on MPApp.vcproj.

**Tip**

We strongly recommend that users set an environment variable called NSPR_ROOT to point to the root directory of their NSPR installation. This environment variable will be referenced below to simplify our examples and to reduce the amount of typing required.

**Figure 4.1. Selecting the Visual C++ Project File**

Visual Studio will open, and the MPApp project will be loaded. The unexpanded class view will appear as shown in Figure 4.2 when Visual Studio first loads.

**Figure 4.2. MPApp Project**
Before proceeding, the program arguments must be set. In Visual Studio 7, this can be performed several ways. For example, right-clicking on the project name in the Solution Explorer brings up the menu shown in Figure 4.3. We are interested in changing the project's properties, so we select the Properties item from the popup menu.

Figure 4.3. Project Menu
Selecting this item opens the “Project Pages” dialog, shown in Figure 4.4. In this window, choose the Debugging item. There will be an empty text entry field under the heading Command Arguments. Here, enter the full paths to the VR Juggler configuration files that will be used to run the torus application. The VJ_BASE_DIR environment variable cannot be used here, unfortunately, so the full path to every file must be used. The following shows the beginning of the program arguments listing `sim.base.config` and `sim.wand.mixin.config`:

**Figure 4.4. Project Settings Dialog**
Note that in this example, the / directory separator is used instead of \ for the program arguments. This is not strictly required in this case, but doing it this way maintains consistency with other examples.

Also under the MPApp project settings, the path to the NSPR headers and libraries may need to be filled in. All the Visual C++ project files shipped with VR Juggler refer to the NSPR installation via the NSPR_ROOT environment variable. If this is not set or cannot be used, the paths must be filled in manually.

Once the program arguments are set up, compile the application. Under the Build menu, choose the Build MPApp item as shown in Figure 4.5.

**Figure 4.5. Build MPApp.exe**
Visual C++ will compile the application, and if you have everything configured properly on your computer, the compiling will complete successfully. Once it is done, execute the MPApp program by choosing the Start item from the Debug menu, shown below in Figure 4.6. For the remainder of this book, we will explain how to run applications from the command line rather than from the Visual Studio GUI. Readers can follow whichever method they prefer.

**Figure 4.6. Execute MPApp.exe**
Chapter 5. Running a VR Juggler Sample Program

It is important to note that the same VR Juggler application can be run in simulator mode or in a full-scale VR system with no modifications. What does change is the configuration files used when starting the program. In $VJ_BASE_DIR/share/vrjuggler/data/configFiles, you can find many basic configuration files including those for running in simulator using a mouse and keyboard to simulate VR input devices and some example files based on those used for the old VRAC C2 system. In the directory, you will see some files with names containing “mixin”. These are special files that provide a specific capability not necessarily needed by all applications. They can be mixed in (hence the name) with other configuration files as needed. The configuration files found in the configFiles directory will be referenced in the examples provided, so be sure you know where they are.

1. Required Reading

Before reading any further, make sure you have already read the instructions on how to install VR Juggler (see Chapter 1, Installing VR Juggler) and on how to configure your environment (see Chapter 2, Environment Variables). That information will not be repeated, and it is assumed that you already know what we mean by VJ_BASE_DIR and LD_LIBRARY_PATH, to name two environment variables. At this point, it is also assumed that you already have compiled an application (MPApp in the case of the examples provided), so you should be sure to have read about how to compile a sample VR Juggler application (in Chapter 4, Compiling a VR Juggler Sample Program) before proceeding.

2. Running an Application on the Desktop

Running on the desktop (or in the so-called “simulator mode”) means that your input is simulated and your display windows may have limited functionality. (By “simulated input”, we mean that input is provided through windows that take keyboard and mouse input and translate that into transformations in the virtual world.) Simulator viewports are limited primarily in that they cannot display stereo graphics. It is important to note that a simulator viewport is a special kind of VR Juggler viewport within a display window. Instead of basing its viewpoint on the head position of one of the users, the viewpoint is controlled by a separate camera that is just another positional device. Within a simulator viewport, VR Juggler draws certain objects to help visualize the environment. For example, the heads of users are represented as blue ellipsoids with gray eyes, and a wand (if present) is drawn as a green pointing device. Besides these common simulator objects, display surfaces can be drawn. These represent projection screens or HMD viewing projections and are drawn as translucent rectangles.

As mentioned, several simulator configuration files are provided with a VR Juggler distribution. These files provide a complete simulation of an immersive environment. Please note that this documentation reflects the state of the configuration files at the time the documentation was written. For more information about the configuration files and how to view or modify the configuration, refer to the VRJConfig Guide. (Using VRJConfig is the best way to find out how a specific configuration file is set up.) The configuration files of interest for simulator mode are as follows:

- **sim.base.config** - The basic configuration file needed by all applications when run in simulator mode. It defines commonly used VR Juggler concepts that are beyond the scope of this particular book. It also defines simulated head movement using the keyboard. For now, it is sufficient to know that it is required to run the sample applications in simulator mode.

  This file also contains the basic simulator display configuration file needed by all applications when run in simulator mode. It defines the display windows where the rendering magic occurs. Two simulator display windows are configured by this file: a small one that is active by default and a larger one that is inactive initially.

---

The VRAC C2 has been replaced, and the files in $VJ_BASE_DIR/share/vrjuggler/data/configFiles are maintained primarily as an example of how to configure a multi-wall projection system. They do not necessarily work, but they can be used as a starting point.
• **sim.analog.wand.mixin.config** - A “mix-in” configuration file that defines simulated analog input using the keyboard. This is only required for applications where analog input is used and needs to be simulated when in simulator mode.

• **sim.analog.mixin.config** - This version of the analog simulator opens its own window. See the previous file (**sim.analog.wand.mixin.config**) for other details.

• **sim.c6displays.mixin.config** - A “mix-in” configuration file that defines the surface displays of VRAC’s C6. This is not required for any application but can be used to test opening multiple display windows (each containing either a surface or a simulator viewport) before running in a multi-screen VR system.

• **sim.digital.glove.mixin.config** - A “mix-in” configuration file that defines simulated digital glove input using the keyboard. This is only required for applications where digital glove input is used and needs to be simulated when in simulator mode.

• **sim.glove.mixin.config** - A “mix-in” configuration file that defines simulated gesture-based glove input using the keyboard. This is only required for applications where gesture-based glove input is used and needs to be simulated when in simulator mode.

• **sim.wand.mixin.config** - A “mix-in” configuration file that defines simulated wand input using the mouse. This is only required for applications where wand input is used and needs to be simulated when in simulator mode.

• **standalone.config** - A configuration file that stands on its own and combines the functionality of **sim.base.config** and **sim.wand.mixin.config**. Note that it uses a single display window for all input.

**Note**

At the time of this writing, this configuration file only works with OpenGL, OpenSG, and Open Scene Graph applications on UNIX and Win32. It will not work with OpenGL Performer at this time.

For the MPApp application, we need the base configuration file and the wand mix-in configuration file.

Now it is time to run the application—finally! Make sure that all your environment variables are set properly before trying to start the application. Once you are ready, specify the name of the application and all the configuration files it needs. An example of this is:

```
% MPApp sim.base.config sim.wand.mixin.config
```

You will notice that no paths are specified for finding the three configuration files. This is intentional to shorten the command line given in the example. In general, you always have to give the full path to the files 2. Beginning users will typically want to reference the example configuration files in `$VJ_BASE_DIR/share/vrjuggler/data/configFiles`. As you get more comfortable with VR Juggler and its configuration system, you may want to make your own modified files and put them in the directory `$HOME/.vjconfig`. To simplify running applications, you may want to make a shell script (or batch file as appropriate) that does all the work of passing configuration files and common command-line arguments. For now, though, use the path `$VJ_BASE_DIR/share/vrjuggler/data/configFiles` for each of the configuration files you pass on the command line.

As the application starts, you will see a plethora of output (more or less depending on how you have `$VPR_DEBUG_NFY_LEVEL`, `$VPR_DEBUG_ALLOW_CATEGORIES`, and `$VPR_DEBUG_DISALLOW_CATEGORIES` set), and then one moderately sized simulator display window will open on the left side of your screen while three blank keyboard input windows open on the right side of your screen. The display window will be titled “SimWindow1”, and the keyboard input windows will be titled “Head Keyboard”, “Sim View Cameras Control” and “Wand Keyboard” (in order from the top of the display to the bottom). Do not

---

2This is being remedied for VR Juggler 2.0 and beyond.
worry that the keyboard windows are black—that is normal. The display window will have an animated blue mesh, a cyan ellipsoid, and a green pointer. The mesh is what you have come to see; the ellipsoid is the user's head; and the pointer is the user's hand. In Figure 5.1, we show what this looks on a RedHat Linux 7.2 desktop for comparison with what you are seeing. Note that the head and wand are only rendered in the simulator windows. They are present because head and wand input are being simulated, and it is typically quite helpful to see the results of that simulated input. To exit the application, press ESC in the window titled “Head Window”.

**Figure 5.1. MPApp running on a Linux desktop with multiple input windows**

With VR Juggler 1.1/2.0, it is possible to use a single window for graphics and for input. To use such a configuration, execute MPApp as follows:

```bash
% MPApp standalone.config
```

This time, only a single window opens, as shown in Figure 5.2. It shows the same graphics as before, but now it is configured to take keyboard and mouse input. To exit, press ESC in the graphics window.

**Figure 5.2. MPApp running on a Linux desktop with one window**
So now you are probably wondering what you can do with this fancy application. Both of the preceding configurations use the same keyboard/mouse mappings; they vary only in which windows accept the keyboard and mouse input. Using the multi-window configuration, head movement is done with the keyboard in “Head Keyboard”; camera movement is done with the keyboard in “Sim View Cameras Control”; and wand movement is done with the keyboard and mouse in “Wand Keyboard”. Using the single-window configuration, all input is done with the keyboard and mouse in “Sim Window”. Note, however, that for the single-window configuration, the camera is attached to the user's head for an over-the-shoulder view, and hence, it does not move separately from the head. For information on how to verify these settings and to view the current configuration, refer to the VRJConfig Guide. The following list of tables provides all the keyboard and mouse controls for the simulator when using these particular configuration files. Note that it is possible to reconfigure the simulator to suit your preferences. This is provided mainly for those who just want something that works now.

**Table 5.1. Moving the simulated head**

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Key Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move head backward</td>
<td>2 on keypad</td>
</tr>
<tr>
<td>Move head left</td>
<td>4 on keypad</td>
</tr>
</tbody>
</table>
### Transformation | Key Press
--- | ---
Move head right | 6 on keypad
Move head forward | 8 on keypad
Move head down | 7 on keypad
Move head up | 9 on keypad
Turn head up | CTRL+2 on keypad
Turn head left | CTRL+4 on keypad
Turn head right | CTRL+6 on keypad
Turn head down | CTRL+8 on keypad
Rotate head clockwise | 1 on keypad
Rotate head counter-clockwise | 3 on keypad

**Table 5.2. Moving the simulated wand**

| Transformation | Mouse Input/Key Press |
--- | --- |
Move wand backward | ALT+move mouse backward |
Move wand forward | ALT+move mouse forward |
Move wand left | CTRL+move mouse left |
Move wand right | CTRL+move mouse right |
Move wand up | CTRL+move mouse forward |
Move wand down | CTRL+move mouse backward |
Rotate wand left | SHIFT+move mouse left |
Rotate wand right | SHIFT+move mouse right |
Rotate wand up | SHIFT+move mouse backward |
Rotate wand down | SHIFT+move mouse forward |
Rotate wand clockwise | Right arrow |
Rotate wand counter-clockwise | Left arrow |
Wand button #1 | Left mouse button |
Wand button #2 | Middle mouse button |
Wand button #3 | Right mouse button |
Wand button #4 | 4 |
Wand button #5 | 5 |
Wand button #6 | 6 |

### Transformation | Key Press
--- | ---
Move camera backward | 2 on keypad
Move camera left | 4 on keypad
Move camera right | 6 on keypad
Move camera forward | 8 on keypad

**Table 5.3. Moving the camera (multi-window configuration only)**
### 3. Running an Application in a VR System

Running an application full-scale in a VR system is more complicated than running in simulator mode. The reason for this is that VR systems tend to differ in configuration and in available hardware. VR Juggler is flexible enough to handle most any configuration you throw at it, but those configurations need to be put together first. Examples of configuration files used in VRAC's C2 system are provided, and they are used in this documentation. It should be noted, however, that for any particular system, custom configuration files will probably have to be written. The idea behind this section is to provide a basic understanding of what is needed to get started with running in a VR system.

The example configuration files in the directory `$VJ_BASE_DIR/share/vrjuggler/data/configFiles` modeled after those used for VRAC's C2 system are as follows:

- **C2.base.config** - The basic configuration file needed by all applications when run in the C2. It defines commonly used VR Juggler concepts that are beyond the scope of this particular book.
- **C2.displays.config** - The basic display configuration file needed to run with all four walls active and rendering stereo graphics. This defines only the four surface displays to be opened.
- **C2.flock.config** - The Ascension Flock of Birds configuration file that defines which bird provides input for the head and for the wand.
- **C2.ibox_buttons.config** - The IBox configuration file that handles the digital wand button inputs.
- **C2.mono.displays.config** - The same configuration as `C2.displays.config` except that the walls are opened to render mono graphics.

Running the application is the same as in simulator mode except that the configuration files given on the command line are different. For example, to run MPApp in the C2 with stereo graphics, the following command would be used:

```bash
% MPApp C2.base.config C2.displays.config C2.flock.config C2.ibox_buttons.config
```
Part II. Application Programming

In this part of the book, we delve into VR application programming using VR Juggler. This is where the technical details come into play. We will make use of many software engineering terms as well as Unified Modeling Language (UML) diagrams to explain concepts and designs.

The chapters in this part of the book build upon each other, so it is important to understand each one in turn. We start with the basic concepts of VR Juggler application objects and the differences between the various graphics APIs supported by VR Juggler. We then review helper classes that are provided for making application authoring easier. This includes a discussion of the most commonly used math classes and functions provided by the Generic Math Template Library (GMTL). The remaining helper class discussion focuses on the methods for getting input in VR Juggler applications through the Gadgeteer input management system.

With these background elements, the subsequent chapters cover application programming using different graphics APIs. We conclude this part of the book with tips for porting existing applications written using GLUT and the CAVElib™ software to VR Juggler.
Chapter 6. Application Basics

In VR Juggler, all applications are written as objects that are handled by the kernel. The objects are known as application objects, and we will use that term frequently throughout this text. Application objects are introduced and explained in this chapter.

1. Application Object Overview

VR Juggler uses the application object to create the VR environment with which the users interact. The application object implements interfaces needed by the VR Juggler virtual platform.

1.1. No main() — “Don’t call me, I’ll call you”

Since VR Juggler applications are objects, developers do not write the traditional main() function. Instead, developers create an application object that implements a pre-defined interface. The VR Juggler kernel controls the application's processing time by calling the object's interface implementation methods.

In traditional programs, the main() function defines the point where the thread of control enters the application. After the main() function is called, the application starts performing any necessary processing. When the operating system (OS) starts the program, it gives the main() function some unit of processing time. After the time unit (quantum) for the process expires, the OS performs what is called a “context switch” to change control to another process. VR Juggler achieves similar functionality but in a slightly different manner.

The application objects correspond to processes in a normal OS. The kernel is the scheduler, and it allocates time to an application by invoking the methods of the application object. Because the kernel has additional information about the resources needed by the applications, it maintains a very strict schedule to define when the application is granted processing time. This is the basis to maintain coherence across the system.

1.2. Application Objects Derive from Base Classes for Specific Graphics APIs

The first step in defining an application object is to implement the basic interfaces defined by the kernel and the Draw Managers. There is a base class for the interface that the kernel expects (vrj::App) and a base class handled by each Draw Manager interface (vrj::PfApp, vrj::GlApp, etc.). See Figure 6.1 for a visual representation of the complete application interface hierarchy. The interface defined in vrj::App specifies methods for initialization, shutdown, and execution of the application. This is the abstract type that is seen by the VR Juggler kernel. The Draw Manager interfaces specified in the vrj::*App classes define the API-specific functions necessary to render the virtual environment. For example, an interface used by a Draw Manager could have functions for drawing the scene and for initializing context-specific information.

Figure 6.1. vrj::App hierarchy
1.3. Writing an Application Means Filling in the Blanks

To implement an application in VR Juggler, developers simply need to “fill in the blanks” of the appropriate interfaces. To simplify this process, there are default implementations of most methods in the interfaces. Hence, the user must only provide implementations for the aspects they want to customize. If an implementation is not provided in the user application object, the default is used, but it is important to know that in most cases, the default implementation does nothing.

Tip

When overriding a virtual method defined by a VR Juggler application class, it is best to call the parent class method implementation before performing any application-specific processing. For example, if a user-defined application object overrides `vrj::App::init()` in the class `userApp`, the method `userApp::init()` should invoke `vrj::App::init()` before performing its own initialization steps.

2. Benefits of Application Objects

As stated earlier, the most common approach for VR application development is one where the application defines the `main()` function. That `main()` function in turn calls library functions when needed. (This is the model followed by software packages such as the CAVElib™ software and the Diverse Toolkit.) The library in this model only executes code when directed to do so by the application. As a result, the application developer is responsible for coordinating the execution of the different VR system components. This can lead to complex applications.

2.1. Allow for Run-Time Changes

As a virtual platform, VR Juggler does not use the model described above because VR Juggler needs to maintain control of the system components. This control is necessary to make changes to the virtual platform at run time. As the controller of the execution, the kernel always knows the current state of the applications, and therefore, it can manage the run-time reconfigurations of the virtual environment safely. With run-time reconfiguration, it is possible to switch applications, start new devices, reconfigure running devices, and send reconfiguration information to the application object.

2.2. Low Coupling

Application objects lead to a robust architecture as a result of low coupling and well-defined inter-object dependencies. The application interface defines the only communication path between the application and the virtual platform, and this allows restriction of inter-object dependencies. This decreased coupling allows changes in the system to be localized, and thus, changes to one object will not affect another unless the interface itself is changed. The result is code that is more robust and more extensible.

Because the application is simply an object, it is possible to load and unload applications dynamically. When the virtual platform initializes, it waits for an application to be passed to it. When the application is given to the VR Juggler kernel at run time, the kernel performs a few initialization steps and then executes the application.

2.3. Allows Implementation Changes

Since applications use a distinct interface to communicate with the virtual platform, changes to the implementation of the virtual platform do not affect the application. Changes could include bug fixes, performance tuning, or new device support.

2.4. Multi-Language Interaction
By treating applications as objects, we can mix programming languages in the VR Juggler kernel. For example, an application object could be written in Python, C#, or even VB.NET, but the VR Juggler kernel (written in standard C++) will still see it as an instance of the abstract interface `vrj::App`. The use of application objects has allowed such extensions to VR Juggler to be written without requiring any changes to VR Juggler.

### 3. VR Juggler Startup

In this section, we describe one way to start VR Juggler. We will use the traditional `main()` function in C++, but this is not the only way to do it. We have written Python applications that start the VR Juggler kernel, and it is possible to write a VR Juggler daemon that loads applications on demand at runtime. In other words, the VR Juggler startup procedure is quite flexible, and we choose to focus on the simplest method here.

#### 3.1. No `main()`—Sort Of

Previously, we explained how VR Juggler applications do not have a `main()` function, but further explanation is required. While it is true that user applications do not have a `main()` function because they are objects, there must still be a `main()` somewhere that starts the system. This is because the operating system uses `main()` as the starting point for all applications. In typical VR Juggler applications, there is a `main()`, but it only starts the VR Juggler kernel and gives the kernel the application to run. It then waits for the kernel to shut down before exiting.

#### 3.2. Structure of a `main()` Function

The following is a typical example of a `main()` function that will start the VR Juggler kernel and hand it an instance of a user application object. The specifics of what is happening in this code are described below.

```cpp
#include <vrj/Kernel/Kernel.h>
#include <simpleApp.h>

int main(int argc, char* argv[]) {

  vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
  simpleApp* app = new simpleApp(); // Create the app object

  kernel->loadConfigFile(...); // Configure the kernel

  kernel->start(); // Start the kernel thread

  kernel->setApplication(app); // Give application to kernel
  kernel->waitForKernelStop(); // Block until kernel stops

  return 0;
}
```

1. This line finds (and may create) the VR Juggler kernel. The kernel reference is stored in the handle so that we can use it later.
2. We instantiate a copy of the user application object (`simpleApp`) here. Notice that we include the header file that defines the `simpleApp` class.
3. This statement represents the code that will be in the `main()` function for passing configuration files to the kernel's `loadConfigFile()` method. These configuration files may come from the command line or from some other source. If reading the files from the command line, it can be as simple as looping through all the arguments and passing each one to the kernel.
4. As a result of this statement, the VR Juggler kernel begins running. It creates a new thread of execution for the
kernel, and the kernel begins its internal processing. From this point on, any changes made reconfigure the kernel. These changes can come in the form of more configuration files or in the form of an application object to execute. At this point, it is important to notice that the kernel knows nothing about the application. Moreover, there is no need for it to know about configuration files yet. This demonstrates how the VR Juggler kernel executes independently from the user application. The kernel will simply work on its own controlling and configuring the system even without an application to run.

This statement finally tells the kernel what application it should run. The method call reconfigures the kernel so that it will now start invoking the application object's member functions. It is at this time that the application is now running in the VR system.

4. Kernel Loop

Before proceeding into application object details, we must understand how VR Juggler calls the application, and we must know what a frame is. In the code above, the statement on line 9 tells the kernel thread to start running. When the kernel begins its execution, it follows the sequence shown in Figure 6.2. The specific methods called are described in more detail in the following section. This diagram will be useful in understanding the order in which the application object methods are invoked.

Figure 6.2. Kernel loop sequence
4.1. Definition of a Frame

The VR Juggler kernel calls each of the methods in the application object based on a strictly scheduled frame of execution. The frame of execution is shown in Figure 6.2; it makes up all the lines within the “while(!quit)” clause.
During the frame of execution, the kernel calls the application methods and performs internal updates (the `updateAllData()` method call). Because the kernel has complete control over the frame, it can make changes at pre-defined “safe” times when the application is not doing any processing. At these times, the kernel can change the virtual platform configuration as long as the interface remains the same.

The frame of execution also serves as a framework for the application. That is, the application can expect that when `preFrame()` is called, the devices have just been updated. Applications can rely upon the system being in well-defined stages of the frame when the kernel invokes the application object's methods.

## 5. Base Application Object Interface

Within this section, we provide a brief overview of the member functions from the base VR Juggler application interface. This interface is defined by `vrj::App`, and the member functions are shown in Figure 6.3. Refer to Figure 6.2 for a visual presentation of the order in which the methods are invoked.

The base interface of the application object defines the following functions:

- `init()`
- `apiInit()`
- `preFrame()`
- `intraFrame()`
- `postFrame()`

As previously described, the VR Juggler kernel calls these functions from its control loop to allocate processing time to them. These functions handle initialization and computation. Other member functions that can be used for reconfiguration, focus control, resetting, and exiting will be covered later in this book.

**Figure 6.3. Application object interface**
5.1. Initialization

The following is a description of the application objects related to the initialization of a VR Juggler application. The order of presentation is the same as the order of execution when the application is executed by the kernel.
5.1.1. vrj::App::init()

The init() method is called by the kernel to initialize any application data. When the kernel prepares to start a new application, it first calls init() to signal the application that it is about to be executed.

5.1.1.1. Timing

This member function is called immediately after the kernel is told to start running the application and before any graphics API handling has been started by VR Juggler.

5.1.1.2. Uses

Typical applications will utilize this method to load data files, create lookup tables, or perform some steps that should be done only once per execution. In other words, this method is the place to perform any pre-processing steps needed by the application to set up its data structures.

5.1.2. vrj::App::apiInit()

This member function is for any graphics API-specific initialization required by the application. Data members that cannot be initialized until after the graphics API is started should be initialized here.

Note

In OpenGL, there is no concept of initializing the API, so this method is normally empty in such applications.

5.1.2.1. Timing

This member function is called after the graphics API has been started but before the kernel frame is started.

5.1.2.2. Uses

In most cases, scene graph loading and other API-specific initialization should be done in this method.

5.2. Frame Functions

Once the application object has been initialized by the VR Juggler kernel, the kernel frame loop begins. Each frame, there are specific application object methods that are invoked, and understanding the timing and potential uses of these methods can improve the functionality of the immersive application. In some cases, it is possible to use these member functions to optimize the application to improve the frame rate and the level of interactivity.

5.2.1. vrj::App::preFrame()

The preFrame() method is called when the system is about to trigger drawing. This is the time that the application object should do any last-minute updates of data based on input device status. It is best to avoid doing any time-consuming computation in this method. The time used in this method contributes to the overall device latency in the system. The devices will not be re-sampled before rendering begins.

5.2.1.1. Timing

This method is called immediately before triggering rendering of the current frame.

5.2.1.2. Uses
In general, this method should be reserved for “last-millisecond” data updates in response to device input (latency-critical code).

5.2.2. vrj::App::intraFrame()

The code in this method executes in parallel with the rendering method. That is, it executes while the current frame is being drawn. This is the place to put any processing that can be done in advance for the next frame. By doing parallel processing in this method, the application can increase its frame rate because drawing and computation can be parallelized. Special care must be taken to ensure that any data being used for rendering does not change while rendering is happening. One method for doing this is buffering. Use of synchronization primitives is not recommended because that technique could lower the frame rate.

5.2.2.1. Timing

This method is invoked after rendering has been triggered but before the rendering has finished.

5.2.2.2. Uses

The primary use of this method is performing time-consuming computations, the results of which can be used in the next frame.

5.2.3. vrj::App::postFrame()

Finally, the postFrame() method is available for final processing at the end of the kernel frame loop. This is a good place to do any data updates that are not dependent upon input data and cannot be overlapped with the rendering process (see the discussion on vrj::App::intraFrame() above).

5.2.3.1. Timing

This method is invoked after rendering has completed but before VR Juggler updates devices and other internal data.

5.2.3.2. Uses

Some possible uses of this method include “cleaning up” after the frame has been rendered or synchronizing with external networking or computational processes.

6. Draw Manager-Specific Application Classes

Beyond the basic methods common to all applications, there are methods that are specific to a given Draw Manager. The application classes are extended for each of the specific Draw Managers. The graphics API-specific application classes derive from vrj::App and extend this interface further. They add extra “hooks” that support the abilities of the specific API.

6.1. OpenGL Application Class

The OpenGL application base class adds several methods to the application interface that allow rendering of OpenGL graphics. The extensions to the base vrj::App class are shown in Figure 6.4. In the following, we describe the method vrj::GlApp::draw(), the most important element of the interface. More details about the vrj::GlApp class are provided in Section 3, “OpenGL Applications”, found in Chapter 8, Writing Applications.

Figure 6.4. vrj::GlApp interface extensions to vrj::App
6.1.1. `vrj::GlApp::draw()`

The “draw function” is called by the OpenGL Draw Manager when it needs to render the current scene in an OpenGL graphics window. It is called for each active OpenGL context.

6.1.2. `vrj::GlApp::getDrawScaleFactor()`

As of VR Juggler 2.0 Alpha 1, applications can specify the units of measure that are the basis for the graphics they render. The default unit of measure is feet (identified by the constant scale factor `gadget::PositionUnitConversion::ConvertToFeet`) to maintain backwards compatibility with the previous VR Juggler semantics. By overriding this method, applications can identify the unit of measure they expect. The default implementation is the following:

```cpp
float vrj::GlApp::getDrawScaleFactor()
{
    return gadget::PositionUnitConversion::ConvertToFeet;
}
```

Overriding this method means changing the rendering scale factor used by the OpenGL Draw Manager. The current list of constants (defined in `gadget/Position/PositionUnitConversion.h`) is as follows:

- `gadget::PositionUnitConversion::ConvertToFeet`
- `gadget::PositionUnitConversion::ConvertToInches`
- `gadget::PositionUnitConversion::ConvertToMeters`
- `gadget::PositionUnitConversion::ConvertToCentimeters`

Because the value returned is simply a scaling factor, user applications can define whatever units they want. Note that internally, VR Juggler is treating all units as *meters*, so the scaling factor converts from meters to the desired units.
6.2. OpenGL Performer Application Class

The OpenGL Performer application base class adds interface functions that deal with the OpenGL Performer scene graph. Some of the interface extensions are shown in Figure 6.5. The following is a description of only two methods in the \texttt{vrj::PfApp} interface. More detailed discussion on this class is provided in Section 4, “OpenGL Performer Applications”, found in Chapter 8, \textit{Writing Applications}.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure6_5}
\caption{vrj::PfApp interface extensions to vrj::App}
\end{figure}
6.2.1. `vrj::PfApp::initScene()`

The `initScene()` member function is called when the application should create the scene graph it will use.
6.2.2. `vrj::PfApp::getScene()`

The `getScene()` member function is called by the Performer Draw Manager when it needs to know what scene graph it should render for the application.
Chapter 7. Helper Classes

Within this chapter, we present information on some helper classes that are provided for use with VR Juggler. These classes are intended to make it easier for application programmers to write their code. Ultimately, we want application programmers to focus more on compelling immersive content and less on the many details that are involved with 3D graphics programming. The classes presented in this chapter focus on mathematical computations and on input from hardware devices. VR Juggler uses the Graphics Math Template Library or GMTL (part of the Generic Graphics Toolkit [http://ggt.sf.net/] software) for mathematical computation. An overview of the most commonly used GMTL data types and operations is presented here. In addition to the GMTL operations, special attention is paid to Gadgeteer, the input system used by VR Juggler, and its device interfaces and device proxies.

1. The \texttt{gmtl::Vec<S, T>} Helper Class

This section is intended to provide an introduction to how the helper class \texttt{gmtl::Vec<S, T>} works and how it can be used in VR Juggler applications. It begins with a high-level description of the classes which forms the necessary basis for understanding them in detail. Then, examples of how to use all the available operations in the interfaces for these classes are provided. It concludes with a description of the internal details of the classes.

1.1. High-Level Description

The class \texttt{gmtl::Vec<S, T>} is designed to work the same way as a mathematical vector, typically of 3 or 4 dimensions. There are predefined vector types that would normally be used in a VR application that are provided for convenience. That is, a \texttt{gmtl::Vec3f} object can be thought of as a vector of the form \( <x, y, z> \). Similarly, a \texttt{gmtl::Vec4f} can be thought of as a vector of the form \( <x, y, z, w> \). An existing understanding of mathematical vectors is sufficient to know how these classes can be used. The question then becomes, how are they used? We will get to that later, and readers who have experience with vectors can skip ahead. If vectors are an unfamiliar topic, it may be convenient to think of these classes as three- and four-element C++ arrays of floats respectively. Most benefits of the vector concept are lost with that simpler idea, however. Therefore, if the reader needs to think of them as arrays, then arrays should probably be used until vectors feel more comfortable. Once the use of vectors seems familiar and straightforward, readers are encouraged to come back and read further.

Vectors are typically used to contain spatial data or something similar. For convenience, however, they can be visualized as a more general-purpose container for numerical data upon which well-defined operations can be performed. There is no need to constrain thinking of them as only holding the coordinates for some point in space or some other limited-scope use. The GMTL vectors use by VR Juggler retain this generality and can be used wherever vectors come in handy.

\texttt{gmtl::Vec3f} and \texttt{gmtl::Vec4f}, as specific implementations of mathematical vectors, hide vector operations on single-precision floating-point numbers (float) behind a simple-to-use interface. For a single vector, the following standard vector operations are available:

- Inversion (changing the sign of all elements)
- Normalization
- Calculation of length
- Multiplication by a scalar
- Division by a scalar
- Conversion to a Performer vector
For two vectors, the following operations can be performed:

- Assignment
- Equality/inequality comparison
- Dot product
- Cross product
- Addition
- Subtraction

Using GMTL vectors should be straightforward if readers understand these operations and keep in mind that `gmtl::Vec3f` and `gmtl::Vec4f` can be thought of at this high level.

### 1.2. Using `gmtl::Vec3f` and `gmtl::Vec4f`

With an understanding of these classes as standard mathematical vectors, it is time to learn how to deal with them at the C++ level. In some cases, the mathematical operators are overloaded to simplify user code; in other cases, a named method must be invoked on an object. Before any of that, however, make sure that the source file includes the `gmtl/Vec.h` header file. From here on, the available operations are presented in the order they were listed in the previous section. We begin with creating the objects and setting their values.

### 1.3. Creating Vectors and Setting Their Values

Before doing anything with vectors, some must be created. The examples here use `gmtl::Vec3f`, but the example is equally applicable to `gmtl::Vec4f`. To create a `gmtl::Vec3f`, use the default constructor which initializes the vector to `<0.0, 0.0, 0.0>`:

```cpp
gmtl::Vec3f vec1;
```

After creating the vector `vec1`, its elements can be assigned values all at once as follows:

```cpp
vec1.set(1.0, 1.5, -1.0);
```

or individually:

```cpp
vec1[0] = 1.0;
vec1[1] = 1.5;
vec1[2] = -1.0;
```

Note that in the last example, the individual elements of the vector can be accessed exactly as with a normal array. To do the above steps all at once when the vector is created, give the element values when declaring the vector:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0);
```

All of the above code has exactly the same results but accomplishes them in different ways. This flexibility is just one of the ways that GMTL vectors are more powerful than C++ arrays (of the same size, of course).

### 1.4. Inversion (Finding the Negative of a Vector)
Once a vector is created, the simplest operation that can be performed on it is finding its inverse. The following code demonstrates just that:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2;
vec2 = -vec1;
```

The vector \( \mathbf{vec2} \) now has the value \<-1.0, -1.5, 1.0>\. That is all there is to it. (Readers interested in details should note that the above does a copy operation to return the negative values.)

### 1.5. Normalization

Normalizing a vector is another simple operation (at the interface level anyway). The following code normalizes a vector:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0);
gmtl::normalize( vec1 );
```

The vector \( \mathbf{vec1} \) is now normalized. Clean and simple.

Besides normalizing a given vector, a vector can be tested to determine if it has already been normalized. This is done as follows (assuming the vector \( \mathbf{vec} \) has already been declared before this point):

```cpp
if ( gmtl::isNormalized( vec1 ) )
{
    // Go here if vec is normalized
}
```

### 1.6. Length Calculation

Part of normalizing a vector requires finding its length first. To get a vector's length, do the following:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0);
float length;
length = gmtl::length( vec1 );
```

In this case, length is assigned the value 2.061553 (or more accurately, the square root of 4.25). Finding the length of a vector appears simple from the programmer's perspective, but it has some hidden costs. Namely, it requires a square root calculation. For optimization purposes, GMTL provides a function called `gmtl::lengthSquared()` that returns the length of the vector without calculating the square root.

### 1.7. Multiplication by a Scalar

The GMTL vector classes provide an easy way to multiply a vector by a scalar. There are several ways to do it depending on what is required. Examples of each method follow.

To multiply a vector by a scalar and store the result in another vector, do the following:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2;
vec2 = 3 * vec1;
```
To multiply a vector by a scalar and store the result in the same vector, do the following:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0);
vec1 *= 3;
```

After this, `vec1` has the value `<3.0, 4.5, -3.0>.

### 1.8. Division by a Scalar

Very similar to multiplying by a scalar, division by scalars is also possible. While the examples are almost identical, they are provided here for clarity.

To divide a vector by a scalar and store the result in another vector, do the following:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2;
vec2 = vec1 / 3;
```

Here, `vec2` gets the value `<0.333333, 0.5, -0.333333>`. Note that the scalar must come after the vector because the operation would not make sense otherwise.

To divide a vector by a scalar and store the result in the same vector, do the following:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0);
vec1 /= 3;
```

After this, `vec1` has the value `<0.333333, 0.5, -0.333333>.

### 1.9. Converting to an OpenGL Performer Vector

SGI's OpenGL Performer likes to work with its own `pfVec3` class, and to facilitate the use of it with `gmtl::Vec3f`, two conversion functions are provided for converting a `gmtl::Vec3f` to a `pfVec3` and vice versa. The first works as follows:

```cpp
gmtl::Vec3f vj_vec;
pfVec3 pf_vec;
// Do stuff to vj_vec...
pf_vec = vrj::GetPfVec(vj_vec);
```

where `vj_vec` is passed by reference for efficiency. (`pf_vec` gets a copy of a `pfVec3`.) To convert a `pfVec3` to a `gmtl::Vec3f`, do the following:

```cpp
pfVec3 pf_vec;
gmtl::Vec3f vj_vec;
// Do stuff to pf_vec...
vj_vec = vrj::GetVjVec(pf_vec);
```
Here again, \texttt{pf\_vec} is passed by reference for efficiency, and \texttt{vj\_vec} gets a copy of a \texttt{gmtl::Vec3f}. Both of these functions are found in the header \texttt{vrj/Draw/Pf/PfUtil.h}.

### 1.10. Assignment

We have already demonstrated vector assignment, though it was not pointed out explicitly. It works just as vector assignment in mathematics. The C++ code that does assignment is as follows:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2;
vec2 = vec1;
```

After the assignment, \texttt{vec2} has the value \texttt{<-1.0, -1.5, 1.0>}. Ta da! Note that this is a copy operation which is the case for all the types of assignments of GMTL vectors.

### 1.11. Equality/Inequality Comparison

To compare the equality of two vectors, there are three available methods (one is just the complement of the other, though):

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
if ( gmtl::isEqual(vec1, vec2) )
{
    // Go here if vec1 and vec2 are equal.
}
```

or

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
if ( vec1 == vec2 )
{
    // Go here if vec1 and vec2 are equal.
}
```

or

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
if ( vec1 != vec2 )
{
    // Go here if vec1 and vec2 are not equal.
}
```

Choose whichever method is most convenient.

### 1.12. Dot Product

Given two vectors, finding the dot product is often needed. GMTL vectors provide a way to do this quickly so that programmers can save themselves the time of typing in the formula over and over. It works as follows:

```cpp
gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
float dot_product;
```
dot_product = gmtl::dot(vec1, vec2);

Now, dot_product has the value 4.0.

1.13. Cross Product

Besides the dot product of two vectors, the cross product is another commonly needed result. It is calculated thusly:

gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0), vec3;
vec3 = gmtl::cross(vec1, vec2);

The result is that vec3 gets a copy of vec1 cross vec2.

1.14. Addition

Adding two vectors can be done one of two ways. The first method returns a resulting vector, and the second method performs the addition and stores the result in the first vector.

gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0), vec3;
vec3 = vec1 + vec2;

Now, vec3 has the value <2.5, 2.5, -2.0>.

gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
vec1 += vec2;

This time, vec1 has the value <2.5, 2.5, -2.0>.

1.15. Subtraction

Subtracting two vectors gives the same options as addition, and while the code is nearly identical, it is provided for the sake of clarity.

gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0), vec3;
vec3 = vec1 - vec2;

Now, vec3 has the value <-0.5, 0.5, 0.0>.

gmtl::Vec3f vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
vec1 -= vec2;

In this case, vec1 has the value <-0.5, 0.5, 0.0>.

1.16. Full Transformation by a Matrix

It is often helpful to apply a transformation to a vector. Transformations are represented by a matrix, so it is necessary to multiply a matrix and a vector. The function gmtl::xform() does this job. For the following example,
assume that there is a \texttt{gmtl::Matrix44f} transformation matrix \texttt{xform\_mat}:

\begin{verbatim}
gmtl::Vec3f vec(1.0, 1.0, 1.0), result\_vec;
gmtl::xform(result\_vec, xform\_mat, vec1);
\end{verbatim}

Depending on the transformations contained within \texttt{xform\_mat}, \texttt{result\_vec} will be transformed fully. The operation as a mathematical equation would be:

\[\text{where } V \text{ and } V' \text{ are vectors and } M \text{ is a } 4\times4 \text{ transformation matrix.}\]

### 1.17. The Gory Details

The details behind \texttt{gmtl::Vec3f} and \texttt{gmtl::Vec4f} really are not all that gory. Internally, they are represented as three- and four-element arrays of floats respectively. Access to these arrays is provided through the member function \texttt{getData()}. For example, this access can be used in the following way:

\begin{verbatim}
gmtl::Vec3f pos(4.0, 1.0982, 10.1241);
glVertex3fv(pos.getData());
\end{verbatim}

Granted, this particular example is rather silly and much slower than just listing the values as the individual arguments to \texttt{glVertex3f()}, but it should get the point across.

In general, the \texttt{getData()} member function should be treated very carefully. Access to it is provided mainly so that operations similar to this example can be performed quickly. An example of abusing access to \texttt{getData()} follows:

\begin{verbatim}
gmtl::Vec4f my\_vec;
my\_vec.getData()[0] = 4.0;
my\_vec.getData()[1] = 1.0982;
my\_vec.getData()[2] = 10.1241;
my\_vec.getData()[3] = 1.0;
\end{verbatim}

Do not do this. It can be confusing to readers of the code who do not necessarily need to know the details of the internal representation. Instead, use one of the methods described above for creating vectors and assigning the elements values.

### 2. The \texttt{gmtl::Matrix44f} Helper Class

This section is intended to provide an introduction into how the helper class \texttt{gmtl::Matrix44f} works and how it can be used in VR Juggler applications. It begins with a high-level description of the class, which forms the necessary basis for understanding it in detail. Then, examples of how to use all the available operations in the interfaces for the class are provided. It concludes with a description of the internal C++ details of \texttt{gmtl::Matrix44f}.

#### 2.1. High-Level Description

Abstractly, \texttt{gmtl::Matrix44f} represents a \(4\times4\) matrix of single-precision floating-point values. The class includes implementations of the standard matrix operations such as transpose, scale, and multiply. More specifically, it is a mechanism to facilitate common matrix operations used in computer graphics, especially those associated with a transform matrix. On the surface, it is nearly identical to a \(4\times4\) C++ array of floats, but there is one crucial difference: \texttt{gmtl::Matrix44f} keeps its internal matrix in column-major order rather than in row-major order. More detail on this is given below, but this is done because OpenGL maintains its internal matrices using the same memory layout. At the conceptual level, this does not matter—it is related only to the matrix representation in the
computer's memory. Access to the elements is still in row-major order. In any case, understanding how C++ multi-
dimensional arrays work means understanding 90% of what there is to know about gmtl::Matrix44f. The class
provides a degree convenience not found with a normal C++ array, especially when programming with OpenGL.
The complications surrounding the gmtl::Matrix44f class are identical to those with OpenGL matrix handling,
and with an understanding of that, then all that is left to learn is the interface of gmtl::Matrix44f.

As a representation of mathematical matrices, gmtl::Matrix44f implements several common operations per#
formed on matrices to relieve the users of some tedious, repetitive effort. The general mathematical operations are:

• Assignment
• Equality/inequality comparison
• Transposing
• Finding the inverse
• Addition
• Subtraction
• Multiplication
• Scaling by a scalar value

The operations well-suited for use with computer graphics are:

• Creating an identity matrix quickly
• Zeroing a matrix in a single step
• Creating an XYZ, a ZYX, or a ZXY Euler rotation matrix
• Constraining rotation about a specific axis or axes
• Making a matrix using direction cosines
• Making a matrix from a quaternion
• Making a rotation transformation matrix about a single axis
• Making a translation transformation matrix
• Making a scale transformation matrix
• Extracting specific transformation information
• Converting to an OpenGL Performer matrix

What is presented here involves some complicated concepts that are far beyond the scope of this documentation.
Without an understanding of matrix math (linear algebra) and an understanding of how transformation matrices
work in OpenGL, this document will not be very useful. It is highly recommended that readers be familiar with these
topics before proceeding. Otherwise, with this high-level description in mind, we now continue on to explain the
gmtl::Matrix44f class at the C++ level.

2.2. Using gmtl::Matrix44f
Keeping the idea of a normal mathematical matrix in mind, we are now ready to look at the C++ use of the `gmtl::Matrix44f` class. Most of the interface is defined using methods, but there are a few cases where mathematical operators have been overloaded to make code easier to read. Before going any further, whenever using a `gmtl::Matrix44f`, make sure to include `gmtl/Matrix.h` first. The operations presented above are now described in detail in the order in which they were listed above. We begin with creating the objects and setting their values.

### 2.3. Creating Matrices and Setting Their Values

Before doing anything with matrices, some must be created first. To create a `gmtl::Matrix44f`, the default constructor can be used. It initializes the matrix to be an identity matrix:

```cpp
gmtl::Matrix44f mat1;
```

After creating this matrix `mat1`, its 16 elements can be assigned values all at once as follows:

```cpp
mat1.set(0.0, 1.0, 2.3, 4.1,
    8.3, 9.0, 2.2, 1.0,
    5.6, 9.9, 9.7, 8.2,
    3.8, 0.9, 2.1, 0.1);
```

or with a float array:

```cpp
float mat_vals[16] =
    {0.0, 8.3, 5.6, 3.8,
    1.0, 9.0, 9.9, 0.9,
    2.3, 2.2, 9.7, 2.1,
    4.1, 1.0, 1.0, 0.1};
mat1.set(mat_vals);
```

Note that when explicitly listing the values with `set()`, they are specified in row-major order. When put into a 16-element array of floats, however, they must be ordered so that they can be copied into the `gmtl::Matrix44f` in column-major order. This is the one exception in the interface where access is column-major (which probably means that the interface has a bug).

To set all the values of a new matrix in one step, they can be given as arguments when declaring the matrix:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1,
    8.3, 9.0, 2.2, 1.0,
    5.6, 9.9, 9.7, 8.2,
    3.8, 0.9, 2.1, 0.1);```

All of the above code has exactly the same results but accomplishes those results in different ways.

To read the elements in a `gmtl::Matrix44f` object, programmers can use either the overloaded `[]` operator or the overloaded `()` operator. The overloaded `[]` operator returns the specified row of the `gmtl::Matrix44f`, and an element in that row can then be read using `[]` again. The code looks exactly the same as with a normal C++ two-dimensional array:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1,
    8.3, 9.0, 2.2, 1.0,
    5.6, 9.9, 9.7, 8.2,
    3.8, 0.9, 2.1, 0.1);```
float val;
val = mat1[3][0];

Here, \( \text{val} \) is assigned the value 3.8. Using the overloaded \((\) operator results in code that looks similar to the way the matrix element would be referenced in mathematics:

\[
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, \\
8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, \\
3.8, 0.9, 2.1, 0.1);
\]

float val;
val = mat1(3, 0);

Again, \( \text{val} \) is assigned the value 3.8. Both of these operations are row-major.

### 2.4. Assignment

Assigning one \( \text{gmtl::Matrix44f} \) to another happens using the normal \( = \) operator as follows:

\[
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2 = mat1;
\]

This makes a \textit{copy} of \( \text{mat1} \) in \( \text{mat2} \) which can be a slow operation.

### 2.5. Equality/Inequality Comparison

To compare the equality of two matrices, there are three available methods (one is just the complement of the other, though):

\[
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
\]

if ( gmtl::isEqual(mat1, mat2) )
{
    // Go here if mat1 and mat2 are equal.
}

or

\[
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
\]

if ( mat1 == mat2 )
{
    // Go here if mat1 and mat2 are equal.
}
or

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
```

if ( mat1 != mat2 )
{
    // Go here if mat1 and mat2 are not equal.
}

Choose whichever method is most convenient.

### 2.6. Transposing

The transpose operation works conceptually as matrix1 = transpose(matrix2). The code is then:

```cpp
gmtl::Matrix44f mat1;
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::transpose(mat1, mat2);
```

The result is stored in mat1. mat2 is passed by reference for efficiency.

### 2.7. Finding the Inverse

The inverse operation works conceptually as matrix1 = inverse(matrix2). The code is then:

```cpp
gmtl::Matrix44f mat1;
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::invert(mat1, mat2);
```

The result is stored in mat1. mat2 is passed by reference for efficiency.

### 2.8. Addition

For the addition operation, the interface is defined so that the sum of two matrices is stored in a third. There are two ways to do addition with `gmtl::Matrix44f`: using the `add()` method or using the overloaded `+` operator. Use of the former is recommended, but the latter can be used if one prefers that style of programming. Examples of both methods follow. The first block of code only declares the `gmtl::Matrix44f` objects.

```cpp
gmtl::Matrix44f mat1;
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat3;
```

Using the `gmtl::add()` function:

```cpp
gmtl::add(mat3, mat1, mat2);
```
Using the overloaded + operator:

\[ \text{mat3} = \text{mat1} + \text{mat2}; \]

The result is stored (via a copy) in \text{mat3}.

### 2.9. Subtraction

For the subtraction operation, the interface is defined so that the difference of two matrices is stored in a third. There are two ways to do subtraction with \texttt{gmtl::Matrix44f}: using the \texttt{sub()} method or using the overloaded - operator. It is recommended that developers use the former, but the latter can be used for stylistic purposes. Examples of both methods follow. The first block of code only declares the \texttt{gmtl::Matrix44f} objects.

\[
\begin{align*}
gmtl::Matrix44f \ &\text{mat1}(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
&\quad 5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1); \\
gmtl::Matrix44f \ &\text{mat2}(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
&\quad 5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1); \\
gmtl::Matrix44f \ &\text{mat3};
\end{align*}
\]

Using the \texttt{gmtl::sub()} method:

\[
\begin{align*}
gmtl::\text{sub(mat3, mat1, mat2);} \\
\end{align*}
\]

Using the overloaded - operator:

\[
\begin{align*}
\text{mat3} = \text{mat1} - \text{mat2};
\end{align*}
\]

The result is stored (via a copy) in \text{mat3}.

### 2.10. Multiplication

As in the case of addition and subtraction, the multiplication interface is defined so that the product of two matrices is stored in a third. This is likely to be the operation used most often since transformation matrices are constructed through multiplication of different transforms. For normal matrix multiplication, there are two ways to do multiplication with \texttt{gmtl::Matrix44f}: using the \texttt{gmtl::mult()} function or using the overloaded * operator. We recommend the use of the \texttt{gmtl::mult()} function but the overloaded * operator can be used by those who prefer that style of programming. Examples of both methods follow. The first block of code only declares the \texttt{gmtl::Matrix44f} objects.

\[
\begin{align*}
gmtl::Matrix44f \ &\text{mat1}(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0, \\
&\quad 5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1); \\
gmtl::Matrix44f \ &\text{mat2}(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
&\quad 5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1); \\
gmtl::Matrix44f \ &\text{mat3};
\end{align*}
\]

Using the \texttt{gmtl::mult()} function:

\[
\begin{align*}
gmtl::\text{mult(mat3, mat1, mat2);} \\
\end{align*}
\]

Using the overloaded * operator:

\[
\begin{align*}
\text{mat3} = \text{mat1} * \text{mat2};
\end{align*}
\]
The result is stored (via a copy) in mat3.

There are two more multiplication operations provided that help in handling the order of the matrices when they are multiplied. These two extra operations do post-multiplication and pre-multiplication of two matrices. An example of post-multiplication is:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::postMult(mat1, mat2);
```

Conceptually, the operation is \( \text{mat1} = \text{mat1} \times \text{mat2} \) so that the second matrix (\text{mat2}) comes as the second factor. The same result can be achieved using the overloaded *= operator:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
mat1 *= mat2;
```

An example of pre-multiplication is:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::Matrix44f mat2(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::preMult(mat1, mat2);
```

Here, the conceptual operation is \( \text{mat1} = \text{mat2} \times \text{mat1} \) so that the second matrix (\text{mat2}) comes as the first factor. In both cases, the result of the multiplication is stored in \text{mat1}.

### 2.11. Scaling by a Scalar Value

Scaling the values of a matrix by a scalar value can be done using two different methods: the \text{setScale()} method or the overloaded \* and / operators that take a single scalar value and returns \text{gmtl::Matrix44f}. As with the preceding operations, we recommend the use of the former, but the latter is available for those who want it. Examples of both methods follow. First, using the \text{gmtl::setScale()} function works as:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::setScale(mat1, 3.0);
```

### 2.12. Making an Identity Matrix Quickly

In computer graphics, an identity matrix is often needed when performing transformations. Because of this, \text{gmtl::Matrix44f} provides a method for converting a matrix into an identity matrix in a single step (at the user code level anyway):

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
```
2.13. Zeroing a Matrix in a Single Step

Before using a matrix, it is often helpful to zero it out to ensure that there is no pollution from previous use. With a `gmtl::Matrix44f`, this can be done in one step:

```cpp
gmtl::Matrix44f mat1(0.0, 1.0, 2.3, 4.1, 8.3, 9.0, 2.2, 1.0,
                      5.6, 9.9, 9.7, 8.2, 3.8, 0.9, 2.1, 0.1);
gmtl::zero(mat1);
```

The result is that all elements of `mat1` are now 0.0.

2.14. Making an XYZ, a ZYX, or a ZXY Euler Rotation Matrix

All the rotation information for a transform can be contained in a single matrix using the methods for making an XYZ, a ZYX, or a ZXY Euler matrix. Code for all three follows:

```cpp
vrj::Matrix mat1;
float x_rot = 0.4, y_rot = 0.541, z_rot = 0.14221;
gmtl::setRot(mat1, gmtl::EulerAngleXYZf(x_rot, y_rot, z_rot));
gmtl::setRot(mat1, gmtl::EulerAngleZYXf(z_rot, y_rot, x_rot));
gmtl::setRot(mat1, gmtl::EulerAngleZXYf(z_rot, x_rot, y_rot));
```

In every case, the matrix is zeroed before the rotation transformation is stored. The result of the above code is that `mat1` is a ZXY Euler rotation matrix. The previous two operations are destroyed.

2.15. Making a Translation Transformation Matrix

To make a translation matrix, there are two methods with each having two different types of arguments specifying the translation. The first makes a matrix with only the given translation (all other transformation information is destroyed):

```cpp
gmtl::Matrix44f mat;
gmtl::Vec3f trans(4.0, -4.231, 1.0);
mat = gmtl::makeTrans<gmtl::Matrix44f>(trans);
```

To change the translation of a transformation matrix without completely obliterating all other transformations, use the following instead:

```cpp
gmtl::Vec3f trans(4.0, -4.231, 1.0);
gmtl::setTrans(mat, trans);
```

2.16. Making a Scale Transformation Matrix
To make a transformation matrix that only scales, a simple method is provided. It works as follows:

```cpp
gmtl::Matrix44f mat;
gmtl::Vec3f scale( 1.5, 1.5, 1.5 );
mat = gmtl::makeScale<Matrix44f>(scale);
```

The result is that mat is a transformation matrix that will perform a scale operation. In this specific case, the scaling happens uniformly for x, y, and z.

### 2.17. Extracting Specific Transformation Information

Finally, methods are provided for extracting transformations from a given matrix. The individual rotations and the translation can be read. For the following examples, assume that `mat` is a `gmtl::Matrix44f` object representing arbitrary translation, rotation, and scaling transformations. To get the Z-axis rotation information (an Euler angle), use the following:

```cpp
float z_rot = (gmtl::makeRot<gmtl::EulerAngleXYZf>(mat))[2];
```

The value return is in radians. We can also get the X-axis rotation.

```cpp
float x_rot = (gmtl::makeRot<gmtl::EulerAngleXYZf>(mat))[0];
```

Getting translations is even simpler because translations are collected into a single vector easily.

```cpp
gmtl::Vec3f trans;
gmtl::setTrans(trans, mat);
```

After this, the translation in `mat` is stored in `trans`. The same can be done with a `gmtl::Vec4f` instead of the `gmtl::Vec3f`.

### 2.18. Converting to an OpenGL Performer Matrix

SGI's OpenGL Performer likes to work with its own `pfMatrix` class, and to facilitate the use of it with `gmtl::Matrix44f`, two conversion functions are provided for making conversions. The first works as follows:

```cpp
gmtl::Matrix44f vj_mat;
pfMatrix pf_mat;
// Perform operations on vj_mat...
pf_mat = vrj::GetPfMatrix(vj_mat);
```

where `vj_mat` is passed by reference for efficiency. (`pf_mat` gets a copy of a `pfMatrix` which is a slow operation.) To convert a `pfMatrix` to a `gmtl::Matrix44f`, do the following:

```cpp
pfMatrix pf_mat;
gmtl::Matrix44f vj_mat;
// Perform operations on pf_mat...
vj_mat = vrj::GetVjMatrix(pf_mat);
```

Here again, `pf_mat` is passed by reference for efficiency, and `vj_mat` gets a copy of a `gmtl::Matrix44f`. 

54
Both of these functions are found in the header `vrj/Draw/Pf/PfUtil.h`.

### 2.19. The Gory Details

Now it is time for the really nasty part. Reading this could cause difficulty in understanding the overwhelming amount of information just presented. Do not read any further unless you absolutely have to or you just like to confuse yourself.

C, C++, and mathematics use matrices in row-major order. Access indices are shown in Table 7.1

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>--- Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>--- Array</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>--- Array</td>
</tr>
<tr>
<td>(3,0)</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>--- Array</td>
</tr>
</tbody>
</table>

OpenGL ordering specifies that the matrix has to be column-major in memory. Thus, to provide programmers with a way to pass a transformation matrix to OpenGL in one step (via `glMultMatrixf()`), the `gmtl::Matrix44f` class maintains its internal matrix in column-major order. Note that in the following table, the given indices are what the cells have to be called in C/C++ notation because we are putting them back to back. This is illustrated in Table 7.2.

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>(1,0)</th>
<th>(2,0)</th>
<th>(3,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
<td>(2,1)</td>
<td>(3,1)</td>
</tr>
<tr>
<td>(0,2)</td>
<td>(1,2)</td>
<td>(2,2)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,3)</td>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>

^ ^ ^ ^

Array Array Array Array

As mentioned, all of this is done so that a given `gmtl::Matrix44f` that acts as a full transformation matrix can be passed to OpenGL directly (more or less). For example, with a given `gmtl::Matrix44f` object `mat` upon which painstaking transformations have been performed, the following can be done:

`glMultMatrixf(mat.getData());`

That could not be simpler. All the transformation efforts have culminated into one statement.

For further information, the best possible source of information, especially for this class, is the header file. Read it; understand it; love it.

### 3. The `gadget::DeviceInterface<T>` Helper Class

The concept of device interfaces in VR Juggler applications is something that often causes confusion for new users. Two object-oriented design patterns are combined by `gadget::DeviceInterface<T>`: smart pointers and proxies. Within this section, we aim to explain Gadgeteer device interfaces clearly and simply. We begin with a high-level description and then move right into using the class.
3.1. High-Level Description

Physical devices are never accessed directly by VR Juggler applications. Instead, the applications are granted access to the device through a proxy. A proxy is nothing more than an intermediary who forwards information between two parties. In this case, the two parties are a VR Juggler application and an input device. The application makes requests on the input device through the proxy.

The (template) class `gadget::DeviceInterface<T>` is designed to be a wrapper class around the proxies. Applications could use the proxy classes directly, but `gadget::DeviceInterface<T>` and its type-specific instances (`gadget::PositionInterface`, `gadget::DigitalInterface`, etc.) simplify use of the proxy object they contain. Thus, typical VR Juggler application objects will have one or more device interface member variables.

Note

The class `gadget::DeviceInterface<T>` is a templated class based on the proxy type it wraps. Examples of instantiations of `gadget::DeviceInterface<T>` are `gadget::DeviceInterface<gadget::PositionProxy>` and `gadget::DeviceInterface<gadget::KeyboardProxy>`. Typedefs such as `gadget::PositionInterface` (another name for `gadget::DeviceInterface<gadget::PositionProxy>`) make the code more readable.

In the application object, a device interface member variable is used as a smart pointer to the proxy. In C++, a smart pointer is not usually an actual object pointer. Instead, the class acting as a smart pointer overloads the dereference operator `->` so that a special action can be taken when the “pointer” is dereferenced. The dereference operator is just another operator like the addition and subtraction operators, and overloading the dereference operator allows some “magic” to occur behind the scenes. On the surface, the code looks exactly the same as a normal pointer dereference, and in most cases, people reading and writing the code can think of the smart pointer as a standard pointer. It may also be convenient to think of a smart pointer as a handle.

With that background, we can move on to explain how `gadget::DeviceInterface<T>` uses these concepts. First, know that `gadget::DeviceInterface<T>` is a base class for all other device interface classes such as digital interfaces (wand buttons), position interfaces (wands, a tracked user's head), etc. In user code, there will be instances of objects such as `gadget::DigitalInterface`, `gadget::PositionInterface`, etc. Once they are properly initialized, device interface objects (whatever their types may be) will act as smart pointers to the actual Gadgeteer device proxy objects they wrap.

All the instances of `gadget::DeviceInterface<T>` encapsulate a pointer to a Gadgeteer device proxy object. (Remember that these proxy objects act as an intermediary between the application and an input device.) The sub-classes also overload the dereference operator `->` which allows them to act as smart pointers. The dereference operator on a device interface object gives access to the object's hidden proxy pointer. With that access, the methods of the encapsulated proxy object can be invoked, usually to read data. The end result is that user applications get access to the proxy objects they need but through a simpler interface than using the proxies directly.

At this point, it is perfectly reasonable to wonder why Gadgeteer uses a concept that requires all sorts of documentation and explanation. The extra effort is worth it because it allows Gadgeteer to hide the actual type of the device being used. There is no need to know that some specific VR system uses a wireless mouse connected to a PC reading bytes from a PS/2 port that represent button presses. All that matters is knowing which buttons are pressed at a given instant. The class `gadget::DigitalInterface` gives exactly that information, and it quietly hides the messiness of dealing with that crazy mouse, its ugly driver, and its overly complex protocol.

3.2. Using `gadget::DeviceInterface<T>`

As noted above, VR Juggler applications do not usually use `gadget::DeviceInterface<T>` directly. Instead, the typedefs for specific instantiations of `gadget::DeviceInterface<T>` mentioned above will be used. Within this section, we will refer to instantiations of `gadget::DeviceInterface<T>` as “device interfaces.”
The high-level description has already made use of this convention.

Before using a device interface, some objects must be declared. Programmers must choose the type that is appropriate for the type of devices relevant to a given application. All device interface objects must be initialized in the application object's `init()` method. Each device interface type has a method called `init()`. This method takes a single string argument naming the proxy to which the interface will connect. Example names are "VJHead", "VJWand", "VJButton0", and "VJAccelerate". These are all symbolic names specified in VR Juggler configuration files. This makes them easier to remember, and it also contributes to hiding the details about the physical device. With this system, no one needs to care how transformation information from the user's head is generated. Gadgeteer cares, but there is no need for it to tell anyone else. All developers care about is the head transformation matrix. An example of initializing a `gadget::PositionInterface` that connects with the user head proxy is:

```cpp
gadget::PositionInterface head;
head.init("VJHead");
```

Remember that this is to be done in an application object's `init()` method. The actual object used would be a member variable of the application class. Note that here, the normal syntax for calling the method of a C++ object is used rather than using the dereference operator. Until it is initialized, the device interface object cannot act as a smart pointer.

Once device interface objects are all initialized and ready to use, it is time to start using them as smart pointers. This is best part! VR Juggler and Gadgeteer are already working hard in the background to update device proxies, and the application is free to access them. (It is usually best to reference them in the `preFrame()` method, but this may not necessarily be true for all proxies.) Continuing with our example of a `gadget::PositionInterface` to the user head proxy, the following code shows how to read the transformation matrix for the user's head:

```cpp
gmtl::Matrix44f head_mat;
head_mat = head->getData();
```

But wait, that was easy! Believe it or not, the code really is that simple. Simply use the overloaded dereference operator to get access to the position proxy object hidden in `gadget::PositionInterface` to read data from the proxy. Of course, we have not explained the `getData()` method at all yet. That comes from the position proxy class, and that is documented elsewhere.

### 3.3. The Gory Details

What is truly amazing about Gadgeteer device interfaces is, despite their seeming complexity, there is really nothing to them. Trying to trace through the source code is a little tricky, but conceptually, it is all about pointers. Keep in mind that all this documentation was written using nothing more than the Gadgeteer header files as a reference.

As mentioned, the class `gadget::DeviceInterface<T>` is a base class for all the specific types of device interfaces such as positional interfaces, digital interfaces, and analog interfaces. This class maintains the name of the proxy and the proxy index, it provides the all-important `init()` method, and it overloads the dereference operator. C++ templates then handle the different type instantiations for the different Gadgeteer proxy types.

Instantiations of `gadget::DeviceInterface<T>` are used to provide the wrapper to a specific type of proxy. They each contain a pointer to a proxy object of the same conceptual type (positional, digital, and so on). Regardless of the specific instantiations of `gadget::DeviceInterface<T>`, they all return a pointer to their contained proxy so that user code can get the current data from the proxy.

The beauty of it all is that the proxy object being pointed to by the device interface can be changed without affecting the execution of the user application. In other words, the proxies can be changed at run time to point to different *physical* devices. All the while, the user code is still using the smart pointer interface and getting data of some sort. This flexibility is one of the most important features of Gadgeteer, and it is important to understand.
4. The gadget::Proxy Helper Class

This whole proxy scheme can be confusing. We admit that it makes the learning curve for VR Juggler a little steeper, but once you get it, you will know it all. An alternate title for this section is "Horton Hears a Proxy." In this case, Horton is VR Juggler (it is rather elephant-like at times), and the complexity of dealing with these ethereal, ubiquitous proxies causes VR Juggler to take a lot of guff. This section presents the gadget::Proxy class, the base class for the input proxies, making it the one that is used the most. It should be noted, however, that the concept is spreading to other parts of VR Juggler because it is so useful. While this is only the introduction, we will give you the moral of the story now: proxies are important concepts, and you should not step on them.

4.1. High-Level Description

The class gadget::Proxy is the base class for all the proxies in the Gadgeteer Input Manager. A better name would be gadget::InputProxy, and it may help to think of it with that name. As a programmer of VR Juggler applications, knowledge of such proxies does not have to be terribly in-depth. The fact is, most VR Juggler programmers will probably never need to know more about a specific device proxy's interface than the return type of its getData() method. Most of the apparent complexity in the specific device proxy classes is only important to Gadgeteer's internal maintenance of the active proxies.

That said, this section is relatively short. As a programmer, the important thing to know is that a proxy is a pointer to a physical device. Application programmers use the higher level device interface as the mechanism to read data in some form from the device. The device interface encapsulates some type of proxy that in turn points to an input device. That device can be a wand, a keyboard, a light sensor, or a home-brewed device that reads some input and returns it to Gadgeteer in a meaningful way. That is a lot of indirection, but it makes the handling of physical devices by Gadgeteer incredibly powerful. Most importantly, it prevents VR Juggler applications from being tied to specific hardware devices.

4.2. Using gadget::Proxy

To be blunt, application programmers do not use gadget::Proxy. Instead, access to a subclass of gadget::Proxy is given through a device interface acting as a smart pointer. The getData() method of that subclass is used. That method is the window into the soul of an input device. The device interface allows calling getData() for the specific proxy object it encapsulates, and the current state of the device pointed to by the proxy is returned.

Therefore, what must be known is the return type of the specific proxy to which access is granted through the device interface. The naming conventions for the proxies and their interfaces makes it relatively simple to determine which proxy object is being encapsulated by which device interface. For example, a gadget::DigitalInterface holds a gadget::DigitalProxy pointer. In that case, refer to the documentation for the gadget::DigitalProxy class and find the return type of getData() (int in this case). The proxy header files have the information, too. These are located in $GADGET_BASE_DIR/include/gadget/Type. Just search for the getData() methods therein.

4.3. The Gory Details

The gory details of gadget::Proxy and its subclasses are not really relevant to this particular section. The subclasses look complicated, and they can be. It is important to note, however, that the complication is part of the interface used internally by Gadgeteer rather than the interface used by the application programmer. Because of that and because each device proxy class is different, those details will not be addressed here. It is sufficient to deal with getData() alone in applications. Leave the ugliness up to VR Juggler and Gadgeteer; they can handle it.
Chapter 8. Writing Applications

This chapter alone comprises the bulk of information about application development. Each section outlines one area of interest for application developers. For example, there are sections that show how to get input from the system and others that show how to write applications for each of the currently supported graphics APIs. Please note that when writing an application, there will be overlap between these sections. For example, an application that needs input, sound, and OpenGL graphics will be based on concepts from each of the relevant sections.

1. Application Review

Before getting into too much detail, we present this section as a review from earlier chapters. There is no new information here; it is simply a quick overview of the basics of VR Juggler applications.

1.1. Basic Application Information

As described in Chapter 6, Application Basics, all VR Juggler applications derive from a base application object class (vrj::App). This class defines the basic interface that VR Juggler expects from all application objects. This means that when constructing an application, the user-defined application object must inherit from vrj::App or from a Draw Manager-specific application class that has vrj::App as a superclass. For example:

```cpp
class userApp : public vrj::App
{
public:
    init();
    preFrame();
    postFrame();
}
```

This defines a new application class (userApp), instances of which can be used anywhere that VR Juggler expects an application object.

1.2. Draw Manager-Specific Application Classes

A user application does not have to (and in most cases does not) derive from vrj::App directly. In almost all cases, an application class is derived from a Draw Manager-specific application class. For example:

```cpp
class userGlApp : public vrj::GlApp
{
public:
    init();
    preFrame();
    postFrame();
    draw();
}
```

This is an example of an OpenGL application. The application class (userGlApp) has derived directly from the OpenGL Draw Manager-specific vrj::GlApp application base class. This class provides extra definitions in the interface that are custom for OpenGL applications.

2. Getting Input

There are many types of input devices that VR Juggler applications can use including positional, digital, and analog.
All applications share the same processes and concepts for acquiring input from devices. The main thing to remember about getting input in applications is that all VR Juggler applications receive input through device handles managed by `gadget::DeviceInterface<T>` instantiations. There are `gadget::DeviceInterface<T>` instantiations for each type of input data that Gadgeteer can handle. There is one for positional input, one for analog, and so on. They all have very similar interfaces and behave exactly the same. (Refer to Section 3, “The gadget::DeviceInterface<T> Helper Class” and Section 4, “The gadget::Proxy Helper Class” for more information.)

### 2.1. How to Get Input

While there has already been a brief presentation about getting input in an application, we need something more. Since all device interfaces look the same, we will focus on an example of getting positional input. All other types are very similar. We begin with a simple application object skeleton.

```cpp
class myApp : public vrj::App
{
public:
    init();
    preFrame();
private:
    gadget::PositionInterface mWand;
}
```

Note the declaration of the variable `mWand` of type `gadget::PositionInterface`. This is the first addition to an application. Device interfaces are usually member variables of the user application class, as in this example.

```cpp
myApp::init()
{
    mWand.init("NameOfPosDevInConfiguration");
}
```

The device interface has to be told about the device from which it will get data. This is done by calling the device interface object's `init()` method with the symbolic string name of the device. This device name comes from the active configuration. We are now ready to read from the device.

```cpp
gmtl::Matrix44f wand_pos(mWand->getData());
```

The above code shows an example of using the positional device interface in an application. It shows some sample code where the application copies the positional information from a device interface. When it is dereferenced, the device interface figures out what device it points to and returns the data from that device. Again, refer to Section 3, “The gadget::DeviceInterface<T> Helper Class” for more information about using `gadget::DeviceInterface<T>`.

### 2.2. Where to Get Input

In the previous section, we showed how to get input from devices, but we never said where to put the code. The location, surprisingly, is application dependent. There are some very good guidelines regarding where applications should process input. Before explaining them, however, we should review the VR Juggler kernel control loop, presented again in Figure 8.1.

**Figure 8.1. VR Juggler kernel control loop**
This diagram looks complicated, but the key here is the `updateAllData()` call near the bottom of the diagram. This is where the Gadgeteer Input Manager updates all the cached device data that will be used in drawing the next frame. This updated copy is used by all user references to device data until the next update and the end of the next frame of execution.
This means two things:

1. The device data is most fresh in \texttt{vrj::App::preFrame()}, and
2. Any time spent in \texttt{vrj::App::preFrame()} increases the overall system latency.

The first point is important because it means that the copy of the device data with the lowest latency is always available in the \texttt{preFrame()} member function. The second point is equally important because it says why user applications should not waste any time in \texttt{preFrame()}. Any time spent in \texttt{preFrame()} increases system latency and in turn decreases the perceived quality of the environment. Hence, it is crucial to avoid placing computations in \texttt{preFrame()}.

### 2.3. Tutorial: Getting Input

#### Table 8.1. Tutorial Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Simple application that prints the location of the head and the wand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>Understand how to get positional and digital input in a VR Juggler application</td>
</tr>
<tr>
<td>Member functions</td>
<td>• \texttt{vrj::App::init()}</td>
</tr>
<tr>
<td></td>
<td>• \texttt{vrj::App::preFrame()}</td>
</tr>
<tr>
<td>Directory</td>
<td>$VJ_BASE_DIR/share/samples/OGL/simple/simpleInput</td>
</tr>
<tr>
<td>Files</td>
<td>• \texttt{simpleInput.h}</td>
</tr>
<tr>
<td></td>
<td>• \texttt{simpleInput.cpp}</td>
</tr>
</tbody>
</table>

#### 2.3.1. Class Declaration and Data Members

In the following class declaration, note the data members (\texttt{mWand}, \texttt{mHead}, etc.). This application has four device interface member variables: two for positional input (\texttt{mHead} and \texttt{mWand}) and two for digital input (\texttt{mButton0} and \texttt{mButton1}). Each of these member variables will act as a handle to a “real” device from which we will read data in \texttt{preFrame()}.

```cpp
1 class simpleInput : public vrj::GlApp  
2 {  
3     public:  
4         virtual void init();  
5         virtual void preFrame();  
6  
7     public:  
8         gadget::PositionInterface mWand; // Positional interface for Wand position  
9         gadget::PositionInterface mHead; // Positional interface for Head position  
10        gadget::DigitalInterface mButton0; // Digital interface for button 0  
11        gadget::DigitalInterface mButton1; // Digital interface for button 1
```
2.3.2. Initializing the Device Interfaces: \texttt{vrj::App::init()}

The devices are initialized in the \texttt{init()} member function of the application. For each device interface member variable, the application calls the variable's own \texttt{init()} method. The argument passed is the symbolic name of the configured device from which data will be read. From this point on in the application, the member variables are \textit{handles} to the named device.

```cpp
1 virtual void init()
{
    // Initialize devices
    mWand.init("VJWand");
    mHead.init("VJHead");
    mButton0.init("VJButton0");
    mButton1.init("VJButton1");
}
```

2.3.3. Examining the Device Data: \texttt{vrj::App::preFrame()}

The following member function implementation gives an example of how to examine the input data using the device interface member variables.

```cpp
1 virtual void preFrame()
{
    if ( mButton0->getData() )
    {
        std::cout << "Button 0 pressed" << std::endl;
    }
    if( mButton1->getData() )
    {
        std::cout << "Button 1 pressed" << std::endl;
    }
    std::cout << "Wand Buttons:" << std::endl;
    std::cout << " 0:" << mButton0->getData() << std::endl;
    std::cout << " 1:" << mButton1->getData() << std::endl;
    std::cout << std::endl;
    // -- Get Wand matrix --- //
    gmtl::Matrix44f wand_matrix(mWand->getData());
    std::cout << "Wand pos: \n" << wand_matrix << std::endl;
}
```

1 These statements check the status of the two digital buttons and write out a line if the button has been pressed.
2 This writes out the current state of both buttons.
3 The final section prints out the current location of the wand in the VR environment.

3. OpenGL Applications

We can now describe how to write OpenGL applications in VR Juggler. An OpenGL-based VR Juggler application must be derived from \texttt{vrj::GlApp}. This in turn is derived from \texttt{vrj::App}. As was discussed in the application object section, \texttt{vrj::App} defines the base interface that VR Juggler expects of all applications. The \texttt{vrj::GlApp} class extends this interface by adding members that the VR Juggler OpenGL Draw Manager needs to render an OpenGL application correctly.
In Figure 8.2, we see some of the methods added by the `vrj::GlApp` interface: `draw()`, `contextInit()`, and `contextPreDraw()` . These methods deal with OpenGL drawing and managing context-specific data (do not
worry what context data is right now—we cover that in detail later). There are a few other member functions in the interface, but these cover 99% of the issues that most developers face. In the following sections, we will describe how to add OpenGL drawing to an application and how to handle context-specific data. There is a tutorial for each topic.

3.1. Clearing the Color and Depth Buffers

Before describing how to render using OpenGL with VR Juggler, we must cover the more basic topic of clearing the color and depth buffers. We describe this part before explaining how to render graphics because these steps will be common to all VR Juggler applications based on OpenGL.

In VR Juggler 1.1 and beyond, there is support for drawing multiple OpenGL viewports in a single VR Juggler display window. This feature is useful for tiled displays where each viewport renders a specific part of the scene. In order for an OpenGL-based application to work with multiple viewports, the color and depth buffers need to be cleared at the correct times.

In a user application, the method `vrj::GlApp::bufferPreDraw()` is overridden so that it clears the color buffer. For example, the following code clears the color buffer using black:

```cpp
void userApp::bufferPreDraw()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

Now we need to clear the depth buffer. This must be done separately from the color buffer to ensure proper stereo rendering. The depth buffer must be cleared in the application object's `draw()` method, usually as the first step:

```cpp
void userApp::draw()
{
    glClear(GL_DEPTH_BUFFER_BIT);
    // Rendering the scene ...
}
```

3.2. OpenGL Drawing: `vrj::GlApp::draw()`

The most important (and visible) component of most OpenGL applications is the OpenGL drawing. The `vrj::GlApp` class interface defines a `draw()` member function to hold the code for drawing a virtual environment. Hence, any OpenGL drawing calls should be placed in the `vrj::GlApp::draw()` function of the user application.

Adding drawing code to an OpenGL-based VR Juggler application is straightforward. The `draw()` method is called whenever the OpenGL Draw Manager needs to render a view of the virtual world created by the user's application. It is called for each defined OpenGL context, and it may be called multiple times per frame in the case of multi-surface setups and/or stereo configurations. Applications should never rely upon the number of times this member function is called per frame.

When the method is called, the OpenGL model view and projection matrices have been configured correctly to draw the scene. Input devices are guaranteed to be in the same state (position, value, etc.) for each call to the `draw()` method for a given frame.

3.2.1. Recommended Uses

The only code that should execute in this function is calls to OpenGL drawing routines. It is permissible to read from input devices to determine what to draw, but application data members should not be updated in this function.
3.2.2. Possible Misuses

The `draw()` method should not be used to perform any time-consuming computations. Code in this member function should not change the state of any application variables.

3.3. Tutorial: Drawing a Cube with OpenGL

Table 8.2. Tutorial Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Simple OpenGL application that draws a cube in the environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectives</td>
<td>Understand how the <code>draw()</code> member function in <code>vrj::GlApp</code> works; create basic OpenGL-based VR Juggler applications</td>
</tr>
</tbody>
</table>
| Member functions | • `vrj::App::init()`  
• `vrj::GlApp::draw()` |
| Directory    | `$VJ_BASE_DIR/share/samples/OGL/simple/` |
|              | SimpleApp |
| Files        | • `simpleApp.h`  
• `simpleApp.cpp` |

3.3.1. Class Declaration

The following application class is called `simpleApp`. It is derived from `vrj::GlApp` and has custom `init()` and `draw()` methods declared. Note that the application declares several device interface members that are used by the application for getting device data.

```cpp
using namespace vrj;
using namespace gadget;

class simpleApp : public GlApp
{
  public:
    simpleApp();
    virtual void init();
    virtual void draw();

  public:
    PositionInterface mWand;
    PositionInterface mHead;
    DigitalInterface mButton0;

    DigitalInterface mButton1;
};
```

3.3.2. The `draw()` Member Function
The implementation of `draw()` is located in `simpleApp.cpp`. Its job is to draw the environment. A partial implementation follows.

```cpp
1 using namespace gmtl;

void simpleApp::draw()
{
  ...                  
  // Create box offset matrix
  Matrix44f box_offset;
  const EulerAngleXYZf euler_ang(Math::deg2Rad(-90.0f), Math::deg2Rad(0.0f),
                                  Math::deg2Rad(0.0f));
  box_offset = gmtl::makeRot<Matrix44f>(euler_ang);  
  gmtl::setTrans(box_offset, Vec3f(0.0, 1.0f, 0.0f));      
  ...   
  glPushMatrix();     
  // Push on offset
  glMultMatrixf(box_offset.getData());     
  ...   
  drawCube();
  glPopMatrix();
  ...   
}
```

This creates a `gmtl::Matrix44f` object that defines the offset of the cube in the virtual world. The new matrix is pushed onto the OpenGL modelview matrix stack. Finally, a cube is drawn.

In the above, there is no projection code in the function. When the function is called by VR Juggler, the projection matrix has already been set up correctly for the system. All the user application must do is draw the environment; VR Juggler handles the rest. In this example, the `draw()` member function renders a cube at an offset location.

### 3.3.3. Exercise

Change the code so that the cube is drawn at the position of the wand instead of at the `box_offset` location.

### 3.4. Context-Specific Data

Many readers may already be familiar with the specifics of OpenGL. In this section, we provide a very brief introduction to context-specific data within OpenGL, and we proceed to explain how it is used by VR Juggler. Those who are already familiar with context-specific data may skip ahead to Section 3.4.1, “Why it is Needed” or to Section 3.5, “Using Context-Specific Data”.

The OpenGL graphics API operates using a state machine that tracks the current settings and attributes set by the OpenGL code. Each window in which we render using OpenGL has a state machine associated with it. The state machines associated with these windows are referred to as OpenGL rendering contexts.

Each context stores the current state of an OpenGL renderer instance. The state includes the following:

- Current color
- Current shading mode
- Current texture
- Display lists
3.4.1. Why it is Needed

As outlined in the VR Juggler architecture documentation, VR Juggler uses a single memory area for all application data. All threads can see the same memory area and thus share the same copy of all variables. This makes programming normal application code very easy because programmers never have to worry about which thread can see which variables. In the case of context-specific data, however, it presents a problem.

To understand the problem, consider an environment where we use a single display list. That display list is created to draw some object in the scene. We would like to be able to call the display list in our `draw()` method and have it draw the primitives that were captured in it.

The following class skeleton shows an outline of this idea. Do not worry for now that we do not show the code where we allocate the display list—that will be covered later. For now, we see that there is a variable that stores the display list ID (`mDispListId`), and we use it in the `draw()` method.

```cpp
using namespace vrj;

class userApp : public GlApp
{
public:
    draw();
public:
    int mDispListId;
};

userApp::draw()
{
    glCallList(mDispListId);
}
```

Now, imagine that we have a VR system configured that needs more than one display window (a multi-wall projection system, for example). There is a thread for each display, and all the display threads call `draw()` in parallel.

Since all threads share the same copy of the variables, they all use the same `mDispListId` when calling `glCallList()`. This is an error because we call `draw` from multiple windows (that is, multiple OpenGL rendering contexts). The display list ID is not the same in each context. What we need, then, is a way to use a different display list ID depending upon the OpenGL context within which we are currently rendering. Context-specific data comes to the rescue to address this problem.

Context-specific data provides us with a way to get a separate copy of a variable for each OpenGL rendering context. This may sound daunting at first, but VR Juggler manages this special variable so that it appears just as a normal variable. The developer never has to deal with contexts directly. VR Juggler transparently ensures that the correct copy of the variable is being used.

3.4.2. Context-Specific Variables in VR Juggler

The following shows how a context-specific variable appears in a VR Juggler application:

```cpp
using namespace vrj;

class userApp : public GlApp
{
public:
    draw();
public:
    int mDispListId;
};
```

```cpp
userApp::draw()
{
    glCallList(mDispListId);
}
```
GlContextData<int> mDispListId; // Context-specific variable

userApp::draw()
{
    glCallList(*mDispListId);
}

This code looks nearly the same as the previous example. In this case, mDispListId is treated as a pointer, and it has a special template-based type that tells VR Juggler it is context-specific data. When defining a context-specific data member, use the vrj::GlContextData<T> template class and pass the “true” type of the variable to the template definition. From then on, it can be treated as a normal pointer.

Note

The types that are used for context-specific data must provide default constructors. The user cannot directly call the constructor for the data item because VR Juggler has to allocate new items on the fly as new contexts are created.

3.4.3. The Inner Workings of Context-Specific Variables

Curious readers are probably wondering how all of this works. To satisfy any curiosity, we now provide a brief description.

The context data items are allocated using a template-based smart pointer class (vrj::GlContextData<T>). Behind the scenes, VR Juggler keeps a list of currently allocated variables for each context. When the application wants to use a context data item, the smart pointer looks in the list and returns a reference to the correct copy for the current context.

This is all done in a fairly light-weight manner. It all boils down to one memory lookup and a couple of pointer dereferences. Not bad for all the power that it gives.

3.5. Using Context-Specific Data

The VR Juggler OpenGL graphics system is a complex, multi-headed beast. Luckily, developers do not have to understand how the system is working to use it correctly. As long as developers subscribe to several simple rules for allocating and using context data, everything will work fine. This section contains these rules, but it does not describe the rationale behind the rules. Those readers who are interested in the details of why these rules should be followed should please read the subsequent section. It contains much more (excruciating) detail.

3.5.1. The Rules

With the background in how to make a context-specific data member and how to use it in a draw() member function, we can move on to how and where the context-specific data should be allocated. If we want to create a display list, we need to know where we should allocate it.

3.5.1.1. Rule 1: Do not allocate context data in draw()

This is straightforward: do not allocate context data in the draw() member function. There are many reasons for this, but the primary one is that allocation tests would be occurring too many times and at incorrect times. There are better places to allocate context data.

3.5.1.2. Rule 2: Initialize static context data in contextInit()

The place to allocate static context-specific data is the vrj::GlApp::contextInit() member function.
“Static” context data refers to context data that does not change during the application’s execution. An example of static context data would be a display list to render an object model that is preloaded by the application and never changes. It is static because the display list only has to be generated once for each context, and the application can generate the display list as soon as it starts execution.

The `contextInit()` member function is called immediately after creation of any new OpenGL contexts. In other words, it is called whenever new windows open. When it is called, the newly created context is active. This method is the perfect place to allocate static context data because it is only called when we have a new context that we need to prepare (and also because that is what it is designed for).

The following code snippet shows a possible use of the application object’s `contextInit()` method:

**Example 8.1. Initializing context-specific data**

```cpp
void userApp::contextInit()
{
    // Allocate context specific data
    (*mDispListId) = glGenLists(1);
    glNewList((*mDispListId), GL_COMPILE);
    glScalef(0.50f, 0.50f, 0.50f);
    // Call func that draws a cube in OpenGL
drawCube();
    glEndList();
    ...
}
```

This shows the normal way that display lists should be allocated in VR Juggler. Allocate the display list, store it to a context-specific data member, and then fill the display list. Texture objects and other types of context-specific data are created in exactly the same manner.

### 3.5.1.3. Rule 3: Allocate and update dynamic context data in `contextPreDraw()`

The place to allocate dynamic context-specific data is the `contextPreDraw()` member function. “Dynamic” context data differs from static context data in that dynamic data may change during the application’s execution. An example of dynamic data would be a display list for rendering an object from a data set that changes as the application executes. This requires dynamic context data because the display list has to be regenerated every time the application changes the data set.

Consider also the following example. While running an application, the user requests to load a new model from a file. After the model data is loaded, it may be best to put the drawing functions into a fresh display list for rendering the model. In this case, `vrj::GlApp::contextInit()` cannot be used because it is only called when a new context is created. Here, all the windows have already been created. What we need, then, is a callback that is called once per existing context so that we can add and change the context-specific data. That is what `contextPreDraw()` does. It is called once per context for each VR Juggler frame with the current context active.

Please notice, however, that since this method is called often and is called in performance-critical areas, you should not do much work in it. Any time taken by this method directly decreases the draw performance of the application. In most cases, we recommend trying to make the function have a very simple early exit clause such as in the following example. This makes the average cost only that of a single comparison operation.

```cpp
void userApp::contextInit()
{
    if (have work to do)
    {
        // Do it
    }
}
3.6. Context-Specific Data Details

Within this section, we provide the details of context-specific data in VR Juggler and justify the rules presented in the previous section.

Rule 1 says that context-specific data should not be allocated in an application object's `draw()` method. We have already stated that the main reason is that `draw()` is called too many times, and it is called at the wrong time for allocation of context-specific data. To be more specific, the `draw()` method is called for each surface, or for each eye, every frame. Static context-specific data only needs to be allocated when a new window is opened. (Dynamic context-specific data is handled separately.)

3.7. Tutorial: Drawing a Cube using OpenGL Display Lists

Table 8.3. Tutorial Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Drawing a cube using a display list in the <code>draw()</code> member function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectives</td>
<td>Understand how to use context-specific data in an application</td>
</tr>
<tr>
<td>Member functions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vrj::App::init()</td>
</tr>
<tr>
<td></td>
<td>vrj::GlApp::contextInit()</td>
</tr>
<tr>
<td></td>
<td>vrj::GlApp::draw()</td>
</tr>
<tr>
<td>Directory</td>
<td>$VJ_BASE_DIR/share/samples/OGL/simple/contextApp</td>
</tr>
</tbody>
</table>
3.7.1. Class Declaration and Data Members

The following code example shows the basics of declaring the class interface and data members for an application that will use context-specific data. This is an extension of the simple OpenGL application presented in Section 3.3, “Tutorial: Drawing a Cube with OpenGL”. Note the addition of the contextInit() declaration and the use of the context-specific data member `mCubeDlId`.

```cpp
using namespace vrj;

class contextApp : public GlApp
{
  public:
    contextApp() {};
    virtual void init();
    virtual void contextInit();
    virtual void draw();
  ...
  public:
    // Id of the cube display list
    GlContextData<GLuint> mCubeDlId;
  ... 
};
```

3.7.2. The contextInit() Member Function

We now show the implementation of `contextApp::contextInit()`. Here the display list is created and stored using context-specific data. Recall Example 8.1, presented in Section 3.5, “Using Context-Specific Data”. That example was based on this tutorial application.

```cpp
void contextApp::contextInit()
{
  // Allocate context specific data
  (*mCubeDlId) = glGenLists(1);
  glNewList((*mCubeDlId), GL_COMPILE);
  glScalef(0.50f, 0.50f, 0.50f);
  drawCube();
  glEndList();
  ...
}
```

3.7.3. The draw() Member Function

Now that we have a display list ID in context-specific data, we can use it in the `draw()` member function. We render the display list by dereferencing the context-specific display list ID.

```cpp
using namespace gmtl;

void contextApp::draw()
{
  ...
}
```
5 // Get Wand matrix
gmtl::Matrix44f wand_matrix(mWand->getData());
...
   glPushMatrix();
   glPushMatrix();
   glMultMatrixf(wand_mat.getData());
   glCallList(*mCubeDlId);
   glPopMatrix();
   ...
   glPopMatrix();
15 }

3.7.4. Exercise

In the tutorial application code, replace the call to `drawAxis()` with a display list call.

4. OpenGL Performer Applications

Programmers familiar with the use of scene graphs may prefer to use that data structure rather than writing OpenGL manually. While VR Juggler does not provide a scene graph of its own, its design allows the use of existing scene graph software. In VR Juggler 1.1 and beyond, the supported scene graphs are OpenGL Performer from SGI, OpenSG, and Open Scene Graph. This section explains how to use OpenGL Performer to write VR Juggler applications.

A Performer-based VR Juggler application must derive from `vrj::PfApp`. Similar to `vrj::GlApp` presented in the previous section, `vrj::PfApp` derives from `vrj::App`. `vrj::PfApp` extends `vrj::App` by adding methods that deal with scene graph initialization and access. Figure 8.4 shows how `vrj::PfApp` fits into the class hierarchy of a Performer-based VR Juggler application.

**Figure 8.4. vrj::PfApp application class**
Two of the methods added to the application interface by \texttt{vrj::PfApp} are \texttt{initScene()} and \texttt{getScene()}.
These are called by the Performer Draw Manager to initialize the application scene graph and to get the root of the scene graph respectively. They must be implemented by the application (they are pure virtual methods within \texttt{vrj::PfApp}). Additional methods will be discussed in this section, but in many cases the default implementations
of these other methods may be used. A simple tutorial application will be provided to illustrate the concepts presented.

### 4.1. Scene Graph Initialization: \texttt{vrj::PfApp::initScene()}

In an application using OpenGL Performer, the scene graph must be initialized before it can be used. The method \texttt{vrj::PfApp::initScene()} is provided for that purpose. Within this method, the root of the application scene graph should be created, and any required models should be loaded and attached to the root in some way. The exact mechanisms for accomplishing this will vary depending on what the application will do.

During the initialization of OpenGL Performer by VR Juggler, \texttt{vrj::PfApp::initScene()} is invoked after the Performer functions \texttt{pfInit()} and \texttt{pfConfig()} but before \texttt{vrj::App::apiInit()}.

### 4.2. Scene Graph Access: \texttt{vrj::PfApp::getScene()}

In order for Performer to render the application scene graph, it must get access to the scene graph root. The method \texttt{vrj::PfApp::getScene()} will be called by the Performer Draw Manager so that it can give the scene graph root node to Performer. Since the job of \texttt{getScene()} is straightforward, its implementation can be very simple. A typical implementation will have a single statement that returns a member variable that holds a pointer to the application scene graph root node.

**Note**

Make sure that the node returned is not a \texttt{pfScene} object. If it is, then lighting will not work.

#### 4.2.1. Possible Misuses

Do not load any models in this member function. This sort of operation should be done within \texttt{initScene()}.

### 4.3. Tutorial: Loading a Model with OpenGL Performer

#### Table 8.4. Tutorial Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Simple OpenGL Performer application that loads a model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectives</td>
<td>Understand how to load a model, add it to a scene graph, and return the root to VR Juggler</td>
</tr>
<tr>
<td>Member functions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• \texttt{vrj::PfApp::initScene()}</td>
</tr>
<tr>
<td></td>
<td>• \texttt{vrj::PfApp::getScene()}</td>
</tr>
<tr>
<td>Directory</td>
<td>$VJ_BASE_DIR/share/samples/Pf/simple/simplePf</td>
</tr>
<tr>
<td>Files</td>
<td>• simplePfApp.h</td>
</tr>
<tr>
<td></td>
<td>• simplePfApp.cpp</td>
</tr>
</tbody>
</table>
4.3.1. Class Declaration

The following application class is called `simplePfApp`. It is derived from `vrj::PfApp` and has custom `initScene()` and `getScene()` methods declared. Note that this application uses `preForkInit()` which will be discussed later. Refer to `simplePfApp.h` for the implementations of `preForkInit()` and `setModel()`.

```cpp
1 class simplePfApp : public vrj::PfApp
2 {
3     public:
4         simplePfApp();
5     virtual ~simplePfApp();
6         virtual void preForkInit();
7         virtual void initScene();
8         virtual pfGroup* getScene();
9         void setModel(std::string modelFile);
10     public:
11         std::string mModelFileName;
12         pfGroup* mLightGroup;
13         pfLightSource* mSun;
14         pfGroup* mRootNode;
15         pfNode* mModelRoot;
16     };
```

4.3.2. The `initScene()` Member Function

The implementation of `initScene()` is in `simplePfApp.cpp`. Within this method, we create the scene graph root node, the lighting node, and load a user-specified model. The implementation follows:

```cpp
1 void simplePfApp::initScene ()
2 {
3     // Allocate all the nodes needed
4     mRootNode = new pfGroup;
5     // Create the SUN light source
6     mLightGroup = new pfGroup;
7     mSun = new pfLightSource;
8     mLightGroup->addChild(mSun);
9     mSun->setPosition(0.3f, 0.0f, 0.3f, 0.0f);
10    mSun->setColor(PFLT_DIFFUSE, 1.0f, 1.0f, 1.0f);
11    mSun->setColor(PFLT_AMBIENT, 0.3f, 0.3f, 0.3f);
12    mSun->setColor(PFLT_SPECULAR, 1.0f, 1.0f, 1.0f);
13    mSun->on();
14
15    // --- LOAD THE MODEL -- //
16    mModelRoot = pfdLoadFile(mModelFileName.c_str());
17
18    // -- CONSTRUCT STATIC STRUCTURE OF SCENE GRAPH -- //
19    mRootNode->addChild(mModelRoot);
20    mRootNode->addChild(mLightGroup);
21 }
```

1 First, the root node is constructed as a `pfGroup` object.
2 Next, some steps are taken to create a light source for the application.
Finally, the model is loaded using `pfdLoadFile()`, and the model scene graph root node is stored in `mModelRoot`. (The model loader must be initialized prior to calling `pfdLoadFile()`. This is done in `preForkInit()`.)

Finally, the model and the light source nodes are added as children of the root.

### 4.3.3. The `getScene()` Member Function

The Performer Draw Manager will call the application’s `getScene()` method to get the root of the scene graph. The implementation of this method can be found in `simplePfApp.h`. The code is as follows:

```cpp
pfGroup* simplePfApp::getScene ()
{
    return mRootNode;
}
```

The simplicity of this method implementation is not limited to the simple tutorial from which it is taken. All Performer-based VR Juggler applications can take advantage of this idiom where the root node is a member variable returned in `getScene()`.

### 4.4. Other `vrj::PfApp` Methods

Besides the two methods discussed so far, there are several other methods in `vrj::PfApp` that extend the basic `vrj::App` interface. Each is discussed in this section.

#### 4.4.1. `preForkInit()`

Prototype:

```cpp
public void preForkInit();
```

This member function allows the user application to do any processing that needs to happen before Performer forks its processes but after `pfInit()` is called. In other words, it is invoked after `pfInit()` but before `pfConfig()`.

#### 4.4.2. `appChanFunc()`

Prototype:

```cpp
public void appChanFunc(pfChannel* chan);
```

This method is called every frame in the application process for each active channel. It is called immediately before rendering (`pfFrame()`).

#### 4.4.3. `configPWin()`

Prototype:

```cpp
public void configPWin(pfPipeWindow* pWin);
```

This method is used to initialize a pipe window. It is called as soon as the pipe window is opened.

#### 4.4.4. `getFrameBufferAttrs()`
Prototype:

```cpp
public std::vector<int> getFrameBufferAttrs();
```

This method returns the needed parameters for the Performer frame buffer. Stereo, double buffering, depth buffering, and RGBA are all requested by default.

### 4.4.5. drawChan()

Prototype:

```cpp
public void drawChan(pfChannel* chan,
                     void* chandata);
```

This is the method called in the channel draw function to do the actual rendering. For most programs, the default behavior of this function is correct. It makes the following calls:

```cpp
chan->clear();
pfDraw();
```

Advanced users may want to override this behavior for complex rendering effects such as overlays or multi-pass rendering. (See the OpenGL Performer manual pages about overriding the draw traversal function.) This function is the draw traversal function but with the projections set correctly for the given displays and eye. Prior to the invocation of this method, `chan` is ready to draw.

### 4.4.6. preDrawChan()

Prototype:

```cpp
public void preDrawChan(pfChannel* chan,
                         void* chandata);
```

This is the function called by the `default drawChan()` member function before clearing the channel and drawing the next frame (`pfFrame()`).

### 4.4.7. postDrawChan()

Prototype:

```cpp
public void postDrawChan(pfChannel* chan,
                         void* chandata);
```

This is the function called by the `default drawChan()` member function after clearing the channel and drawing the next frame (`pfFrame()`).

### 5. OpenSG Applications

This section explains how to use the OpenSG scene graph in a VR Juggler application. OpenSG is an open source scene graph that is available at [www.opensg.org](http://www.opensg.org/).

An OpenSG-based VR Juggler application must derive from `vrj::OpenSGApp`. The `vrj::OpenSGApp` class is derived from the `vrj::GlApp` presented previously, which in turn derives from `vrj::App`. 

```cpp
Writing Applications
```

---

Page 78
vrj::OpenSGApp extends vrj::GlApp by adding methods that deal with scene graph initialization and access. Figure 8.5 shows how vrj::OpenSGApp fits into the class hierarchy of an OpenSG-based VR Juggler application.

**Figure 8.5. vrj::OpenSGApp application class**

The two main application methods for vrj::OpenSGApp VR Juggler applications are initScene() and getSceneRoot(). These are called by the OpenSG application class wrapper to initialize the application scene graph and to get the root of the scene graph respectively. They must be implemented by the application (they are pure virtual methods within vrj::OpenSGApp). The rest of this section gives a more detailed description of these methods and some sample code to illustrate the concepts presented.

**5.1. Scene Graph Initialization: vrj::OpenSGApp::initScene()**

In an application using OpenSG, the scene graph must be initialized before it can be used. The method
vrj::OpenSGApp::initScene() is provided for that purpose. Within this method, the root of the application scene graph should be created, and any required models should be loaded and attached to the root in some way. The exact mechanisms for accomplishing this will vary depending on what the application will do.

During the API initialization, vrj::OpenSGApp::initScene() is invoked. This happens after OSG::osgInit() has been called, so OpenSG should be fully initialized and ready to be used.

5.2. Scene Graph Access: vrj::OpenSGApp::getSceneRoot()

In order for OpenSG to render the application scene graph, it must get access to the scene graph root. The method vrj::OpenSGApp::getSceneRoot() will be called by the OpenSG application class wrapper so that it can get access to the currently active scene graph whenever the wrapper needs to use it (ex. rendering, updating). Since the job of getSceneRoot() is straightforward, its implementation can be very simple. A typical implementation will have a single statement that returns a member variable that holds a pointer to the current scene graph root node.

5.2.1. Possible Misuses

Do not do any CPU-heavy processing in this method. Because this method is called frequently, it should only do the minimum amount of processing necessary to return the root scene graph node. In most cases this method should only be one line of code. See the following code for an example.

```cpp
virtual OSG::NodePtr getSceneRoot()
{
    return mSceneRoot; // Return the root of the graph
}
```

If you need to update the scene graph, you should use either preFrame(), intraFrame(), or postFrame().

5.3. Tutorial: Loading a Model with OpenSG

Table 8.5. Tutorial Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Simple OpenSG application that loads a model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectives</td>
<td>Understand how to load a model, add it to a scene graph, and return the root to VR Juggler</td>
</tr>
<tr>
<td>Member functions</td>
<td></td>
</tr>
<tr>
<td>vrj::OpenSGApp::initScene()</td>
<td></td>
</tr>
<tr>
<td>vrj::OpenSGApp::getScene()</td>
<td></td>
</tr>
<tr>
<td>Directory</td>
<td>$VJ_BASE_DIR/share/vrjuggler/samples/OpenSG/simple/OpenSGNav</td>
</tr>
<tr>
<td>Files</td>
<td></td>
</tr>
<tr>
<td>OpenSGNav.h</td>
<td></td>
</tr>
<tr>
<td>OpenSGNav.cpp</td>
<td></td>
</tr>
</tbody>
</table>

5.3.1. Class Declaration
The following application class is called OpenSGNav. It is derived from vrj::OpenSGApp and has custom initScene(), getSceneRoot(), init(), contextInit(), and preFrame() methods declared. Refer to OpenSGNav.h for the implementation of setModelFileName().

```cpp
class OpenSGNav : public vrj::OpenSGApp
{
public:
    OpenSGNav(vrj::Kernel* kern);
    virtual ~OpenSGNav();
    virtual void init();
    virtual void contextInit();
    virtual void preFrame();

    virtual void initScene();
    virtual OSG::NodePtr getSceneRoot();

    void setModelFileName(std::string filename);

private:
    void initGLState();

private:

    std::string mFileToLoad;
    OSG::NodePtr mRootNode;
    OSG::TransformPtr mSceneScale;
    OSG::NodePtr mSceneRoot;
    OSG::TransformPtr mSceneTransform;
    OSG::NodePtr mModelRoot;
    OSG::NodePtr mLightNode;
    OSG::NodePtr mLightBeacon;

public:
    gadget::PositionInterface mWandPos;
    gadget::DigitalInterface mButton0;
    gadget::DigitalInterface mButton1;
    gadget::DigitalInterface mButton2;
    float velocity;
};
```

### 5.3.2. The initScene() Member Function

The implementation of initScene() is in OpenSGNav.cpp. Within this method, we create the scene graph root node, the lighting node, and load a user-specified model. The implementation follows:

```cpp
void OpenSGNav::initScene()
{
    // Load the model to use
    if (mFileToLoad == std::string("none"))
    {
        mModelRoot = OSG::makeTorus(.5, 2, 16, 16);
    }
    else
    {
        mModelRoot =
            OSG::SceneFileHandler::the().read(mFileToLoad.c_str());
    }
}
```
// --- Light setup --- //
// - Add directional light for scene
// - Create a beacon for it and connect to that beacon
mLightNode = OSG::Node::create();

mLightBeacon = OSG::Node::create();
OSG::DirectionalLightPtr light_core = OSG::DirectionalLight::create();
OSG::TransformPtr light_beacon_core = OSG::Transform::create();

// Setup light beacon
OSG::Matrix light_pos;
light_pos.setTransform(osg::Vec3f(2.0f, 5.0f, 4.0f));

OSG::beginEditCP(light_beacon_core, OSG::Transform::MatrixFieldMask);
light_beacon_core->setMatrix(light_pos);
OSG::endEditCP(light_beacon_core, OSG::Transform::MatrixFieldMask);

OSG::beginEditCP(mLightBeacon);
mLightBeacon->setCore(light_beacon_core);
OSG::endEditCP(mLightBeacon);

// Setup light node
OSG::addRefCP(mLightNode);
OSG::beginEditCP(mLightNode);
mLightNode->setCore(light_core);
mLightNode->addChild(mLightBeacon);
OSG::endEditCP(mLightNode);

OSG::beginEditCP(light_core);
light_core->setAmbient(.9, .8, .8, 1);
light_core->setDiffuse(.6, .6, .6, 1);
light_core->setSpecular(1, 1, 1, 1);
light_core->setDirection(0, 0, 1);
light_core->setBeacon(mLightNode);
OSG::endEditCP(light_core);

// --- Setup Scene -- //
// add the loaded scene to the light node, so that it is lit by the light
OSG::addRefCP(mModelRoot);
OSG::beginEditCP(mLightNode);

mLightNode->addChild(mModelRoot);
OSG::endEditCP(mLightNode);

// create the root part of the scene
mRootNode = OSG::Node::create();

mSceneScale = OSG::Transform::create();
mSceneRoot = OSG::Node::create();

mSceneTransform = OSG::Transform::create();

// Set the root node
OSG::beginEditCP(mRootNode);
mRootNode->setCore(mSceneScale);
mRootNode->addChild(mSceneRoot);
We begin by loading the file set in OpenSGNav::setModelFileName(). If no file name was set, we default to using a simple torus model. The model object is mModelRoot.

Next, we create a node for the light, which we define as a beacon light. The model is added under the light in the scene graph so that it gets lit. Then, the root node for the scene graph is created. This is what will be returned to OpenSG for rendering by OpenSGNav::getSceneRoot(). We also create a root node for the scene. We do this so that we can give it a transformation object without affecting the root of the scene graph. The scene root is added as a child of the root node. Finally, we add the light node as a child of the scene root. Remember that the light node already has the loaded model as a child.

5.3.3. The getSceneRoot() Member Function

The Performer Draw Manager will call the application's getSceneRoot() method to get the root of the scene graph. The implementation of this method can be found in OpenSGNav.h. The code is as follows:

```cpp
OSG::NodePtr OpenSGNav::getSceneRoot()
{
    return mRootNode;
}
```

The simplicity of this method implementation is not limited to the simple tutorial from which it is taken. All OpenSG-based VR Juggler applications can take advantage of this idiom where the root node is a member variable returned in getSceneRoot().
Chapter 9. Porting to VR Juggler from the CAVElib™

In this chapter, we give some methods for porting an application written with the CAVElib™ software to VR Juggler. We explain the process for an OpenGL application. Throughout, we compare and contrast the techniques used by VR Juggler and the CAVElib™ software, and we translate concepts familiar to CAVElib™ programmers into VR Juggler terms.

1. The Initialize, Draw, and Frame Routines

In the CAVElib™, the initialize, draw, and frame routines are known as callbacks implemented with C function pointers. In VR Juggler, the equivalent routines are “called back” using an application object. An application object is a C++ class that defines methods to encapsulate the functionality of the application within a single C++ object.

1.1. In CAVElib™

The following lists the draw, frame, and initialize routines used in the CAVElib™ software.

- Draw: An application's display callback function is defined by passing a function pointer to CAVEDisplay()
- Frame: The frame function is defined with CAVEFrameFunction()
- Init: The initialization callback is defined using CAVEInitApplication()

1.2. In VR Juggler

With VR Juggler, no C function pointers are necessary, but a pointer to an application object must be given to the VR Juggler kernel. As described in earlier sections of this chapter, the first step is to derive a new application class from vrj::GlApp. For more information on application objects, it may be helpful to review Chapter 6, Application Basics. Briefly, the application class definition would appear similar to the following:

```cpp
class MyApplication : public vrj::GlApp
{
   ...
};
```

The draw, frame, and initialize routine concepts in VR Juggler are presented in the following list.

- Draw: An application's display “callback” function is defined by a member function called draw() in the derived class. This is where OpenGL rendering commands such as glBegin(), glVertex(), etc. are placed.
- Frame: Calculations such as navigation, collision, physics, artificial intelligence, etc. are often placed in the frame function. The frame function is split across three member functions:

1. MyApplication::preFrame(), called before draw()
2. MyApplication::intraFrame(), called during draw()
3. MyApplication::postFrame(), called after draw()
• Init: There is an initialization member function for data and an initialization member function for creating context-specific data (display lists, texture objects). The latter is called for each display context in the system. These two member functions are:

1. MyApplication::init(), called once per application startup
2. MyApplication::contextInit(), called once per display context creation

Readers who find some of these concepts unfamiliar are encouraged to read Section 3, “OpenGL Applications”. For information about context-specific data, refer to Section 3.4, “Context-Specific Data”.

2. Getting Input from Devices

Getting input from the hardware devices is conceptually the same, but the implementations are quite different between the CAVElib™ software and VR Juggler.

2.1. In CAVElib™

To get tracking information, the following functions are used:

- CAVEGetPosition(id, pos)
- CAVEGetOrientation(id, orient)
- CAVEGetVector(id, vec)
- CAVEGetSensorPosition(sensor, coords, pos)
- CAVEGetSensorOrientation(sensor, coords, orient)
- CAVEGetSensorVector(sensor, id, vec)

For button input, the following macros are used:

- CAVEBUTTON1, CAVEBUTTON2, CAVEBUTTON3, CAVEBUTTON4, CAVE_JOYSTICK_X, and CAVE_JOYSTICK_Y
- CAVEButtonChange()

2.2. In VR Juggler

To get device input, use the classes derived from gadget::DeviceInterface. They include the following:

- gadget::PositionInterface (for trackers and other positional devices)
- gadget::DigitalInterface (for buttons and other on/off devices)
- gadget::AnalogInterface (for potentiometers and other multi-range data devices)
For more information about the VR Juggler device interfaces, refer to Chapter 7, Helper Classes. A tutorial on getting device input in VR Juggler applications can be found in Section 2, “Getting Input”.

3. Configuration

Configuration of VR Juggler and the CAVElib™ software is very different. The differences are too numerous to list here, but we give a brief overview and a pointer to the documentation that explains configuration of VR Juggler.

3.1. In CAVElib™

All configurable parameters go in a single file called .caverc. The configuration mechanism is proprietary and not usable by external VR system software. In particular, VR Juggler cannot get its configuration information from an existing .caverc file.

3.2. In VR Juggler

Configuration of VR Juggler is much more powerful and flexible than what is used by the CAVElib™ software. As a result, it is also more complex. All configurable parameters could be in one or more files with any names desired. VR Juggler comes with example configuration files that may be found in the directory $VJ_BASE_DIR/share/vrjuggler/data/configFiles.

The VR Juggler configuration system is completely extensible and could be used outside of VR Juggler. Indeed, it could be used outside of any VR paradigm altogether. Refer to the Configuration Guide for more information on configuring VR Juggler.

4. Important Notes

Finally, before we get to the source code, there are some important notes about programming VR Juggler applications in general. Please read these carefully and refer to the indicated chapters for more information as necessary.

4.1. Shared Memory

Unlike the CAVElib™ software, VR Juggler does not have to manage shared memory with other VR Juggler instances. Thus, when writing a VR Juggler application, memory can be created as in a normal, single-threaded C or C++ application.

4.2. OpenGL Context-Specific Data

As a result of the shared memory model described above, VR Juggler has different requirements for context-specific data than the CAVElib™ software. Information such as display lists and texture objects must be managed using context-specific data. A display context is the location to which OpenGL rendering commands draw. Compiled OpenGL commands such as display lists do not get shared across multiple contexts (or windows), and thus, they must be initialized once per display context. In a VR Juggler application, these OpenGL initializations must be placed in vrj::GlApp::contextInit(). It is called once per display context after each context has become active. For a more detailed description of these concepts and a tutorial on how to use them, please refer to Section 3.4, “Context-Specific Data”.

5. Source Code

This final section is the heart of the porting discussion. We present some source code as a means to illustrate how CAVElib™ concepts map to VR Juggler.
5.1. The Form of a Basic CAVElib™ Program

```c
void app_shared_init();
void app_compute_init();
void app_init_gl();
void app_draw();
void app_compute();
void main(int argc, char **argv)
{
    CAVEConfigure(&argc, argv, NULL);
    app_shared_init(argc, argv);
    CAVEInit();
    CAVEInitApplication(app_init_gl, 0);
    CAVEDisplay(app_draw, 0);
    app_compute_init(argc, argv);
    while (!getbutton(ESCKEY))
    {
        app_compute();
    }
    CAVEExit();
}
```

5.2. The Form of a Basic VR Juggler Program

```c
class MyApplication : public vrj::GlApp
{
    public:
    // Data callbacks (Do not put OpenGL code here)
    virtual void init();
    virtual void preFrame();
    virtual void intraFrame();
    virtual void postFrame();
    // OpenGL callbacks (put only OpenGL code here)
    virtual void contextInit();
    virtual void draw();
};
int main(int argc, char* argv[])
{
    // configure kernel with *.config files
    vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
    for(int i=1; i<argc; i++)
    {
        // loading config file passed on command line...
        kernel->loadConfigFile(argv[i]);
    }
    // start the kernel
    kernel->start();
    // set the application for the kernel to run
    MyApplication* application = new MyApplication();
    kernel->setApplication(application);
    // Block until the kernel exits.
    kernel->waitForKernelStop();
    return 0;
}
```
Chapter 10. Porting to VR Juggler from GLUT

In this chapter, we give some methods for porting an application written with GLUT to VR Juggler. Throughout, we compare and contrast the techniques used by VR Juggler and GLUT, and we translate concepts familiar to GLUT programmers into VR Juggler terms.

1. Window Creation and Management

In VR Juggler, window creation is done behind the scenes based on configuration file settings. There are two display types: Surface and Simulator. A Surface Display can be put into three modes: stereo, right eye, or left eye. Most interesting is the stereo mode. Stereo mode requires special hardware to display stereo, and it creates the most immersive experience. A Simulator Display is special because it emulates an active VR system. It can show the all active user head positions and orientation, any active devices such as gloves or wands, and any Surface Displays. The simulator window is nice for debugging tracking systems and for visualizing configured Surface Displays.

2. The Initialize, Draw, and Frame Routines

2.1. In GLUT

In GLUT, the initialize, draw, and frame routines are known as callbacks implemented with C function pointers. In VR juggler, the equivalent routines are called back using an application object. An application object is a C++ class that defines methods to encapsulate the functionality of the application within a single C++ object.

- Draw: OpenGL commands are placed in the draw routine. The callback function is defined by passing a function pointer to glutDisplayFunc().
- Frame: Operations on application data are done within the frame routine. No OpenGL commands are allowed here because the display window is undefined at this point. The frame function is defined with glutIdleFunc(). This function generally does a glutPostRedisplay() to cause the display callback to be executed.
- Init: There is no callback for initialization. Data initialization is done usually before the application starts. Context initialization is done during the first run of the function set with glutDisplayFunc() (once for each window opened).

2.2. In VR Juggler

With VR Juggler, no C function pointers are necessary, but a pointer to an application object must be given to the VR Juggler kernel. As described in earlier sections of this chapter, the first step is to derive a new application class from vrj::GlApp. For more information on application objects, it may be helpful to review Chapter 6, Application Basics. Briefly, the application class definition would appear similar to the following:

```cpp
class MyApplication : public vrj::GlApp
{
    ...
};
```

The draw, frame, and initialize routine concepts in VR Juggler are presented in the following list.
• Draw: An application's display "callback" function is defined by a new member function called `draw()` in the derived class. This is where OpenGL rendering commands such as `glBegin()`, `glVertex()`, etc. are placed.

• Frame: Calculations such as navigation, collision, physics, artificial intelligence, etc. are often placed in the frame function. The frame function is split across three member functions:
  1. `MyApplication::preFrame()`, called before `draw()`
  2. `MyApplication::intraFrame()`, called during `draw()`
  3. `MyApplication::postFrame()`, called after `draw()`

• Init: There is an initialization member function for data and an initialization member function for creating context-specific data (display lists, texture objects). The latter is called for each display context in the system. These two member functions are:
  1. `MyApplication::init()`, called once per application startup
  2. `MyApplication::contextInit()`, called once per display context creation

Readers who find some of these concepts unfamiliar are encouraged to read Section 3, “OpenGL Applications”. For information about context-specific data, refer to Section 3.4, “Context-Specific Data”.

3. Getting Input from Devices

3.1. In GLUT

For keyboard input, the following functions are used:

• `glutKeyboardFunc(OnKeyboardDown)`
• `glutKeyboardUpFunc(OnKeyboardUp)`
• `glutSpecialFunc(OnSpecialKeyboardDown)`
• `glutSpecialUpFunc(OnSpecialKeyboardUp)`

For mouse input, the following functions are used:

• `glutMouseFunc(OnMouseButton)`
• `glutMotionFunc(OnMousePosition)`
• `glutPassiveMotionFunc(OnMousePosition)`

3.2. In VR Juggler
To get device input, use the classes derived from `gadget::DeviceInterface`. They include the following:

- `gadget::PositionInterface` (for trackers and other positional devices)
- `gadget::DigitalInterface` (for buttons and other on/off devices)
- `gadget::AnalogInterface` (for potentiometers and other multi-range data devices)

For more information about the VR Juggler device interfaces, refer to Chapter 7, *Helper Classes*. A tutorial on getting device input in VR Juggler applications can be found in Section 2, “Getting Input”.

### 4. Configuration

Configuration of GLUT applications is quite different than configuration of VR Juggler applications. In particular, VR Juggler is much more dynamic because configurations are maintained as files separate from the application. In GLUT, the configuration must be written into the application somehow. This can lead to very static, hard-coded configurations.

#### 4.1. In GLUT

There is no built-in configuration system. All system settings are coded using the GLUT API.

#### 4.2. In VR Juggler

VR Juggler has a powerful and flexible configuration system. As a result, it is also complex. All configurable parameters could be in one or more files with any names desired. VR Juggler comes with example configuration files that may be found in the directory `$VJ_BASE_DIR/share/vrjuggler/data/configFiles`.

The VR Juggler configuration system is completely extensible and could be used outside of VR Juggler. Indeed, it could be used outside of any VR paradigm altogether. Refer to the *VRJConfig Guide* for more information on configuring VR Juggler.

### 5. Important Notes

Finally, before we get to the source code, there are some important notes about programming VR Juggler applications in general. Please read these carefully and refer to the indicated chapters for more information as necessary.

#### 5.1. Shared Memory

VR Juggler is multi-threaded, and it uses a shared memory model across all threads. Thus, when writing a VR Juggler application, memory can be created as in a normal, single-threaded C or C++ application. VR Juggler is written entirely in C++, and as such, new and delete must be used instead of `malloc()` and `free()`.

#### 5.2. OpenGL Context-Specific Data

As a result of the shared memory model described above, VR Juggler has different requirements for context-specific data than GLUT. Information such as display lists and texture objects must be managed using context-specific data. A display context is the location to which OpenGL rendering commands draw. Compiled OpenGL commands such as display lists do not get shared across multiple contexts (or windows), and thus, they must be initialized once per display context. In a VR Juggler application, these OpenGL initializations must be placed in `vrj::GlApp::contextInit()`. It is called once per display context after each context has become active. For
6. Source Code

This final section is the heart of the porting discussion. We present some source code as a means to illustrate how GLUT concepts map to VR Juggler.

6.1. The Form of a Basic GLUT Program

```c
void main(int argc, char* argv[])
{
    /* initialize the application data here */
    OnApplicationInit();

    /* create a window to render graphics in
     * In VR Juggler, window creation is done for you based on your configuration file
     * settings.
     */
    glutInitWindowSize( 640, 480 );
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE );
    glutCreateWindow( "GLUT application" );

    /* display callbacks.
     * NOTE: the first time OnIdle is called is when you should
     * initialize the display context for each window
     * (doing this is analogous to VR Juggler's
     * vrj::GlApp::contextInit() function)
     */
    glutReshapeFunc( OnReshape );
    glutIdleFunc( OnIdle );
    glutDisplayFunc( OnIdle );

    /* tell glut to not call the keyboard callback repeatedly
     * when holding down a key. (uses edge triggering, like the mouse does)
     */
    glutIgnoreKeyRepeat( 1 );

    /* keyboard callback functions. */
    glutKeyboardFunc( OnKeyboardDown );
    glutKeyboardUpFunc( OnKeyboardUp );
    glutSpecialFunc( OnSpecialKeyboardDown );
    glutSpecialUpFunc( OnSpecialKeyboardUp );

    /* mouse callback functions... */
    glutMouseFunc( OnMouseClick );
    glutMotionFunc( OnMousePos );
    glutPassiveMotionFunc( OnMousePos );

    /* start the application loop, your callbacks will now be called
     * time for glut to sit and spin. In Juggler this is the same as the while(1)
     * (see below)
     */
    glutMainLoop();
}
```

6.2. The Form of a Basic VR Juggler Program

```c
class MyApplication : public vrj::GlApp
```


```cpp
public:
  // Data callbacks (Do not put OpenGL code here)
  virtual void init();
  virtual void preFrame();
  virtual void intraFrame();
  virtual void postFrame();

  // OpenGL callbacks (put only OpenGL code here)
  virtual void contextInit();
  virtual void draw();

};

int main(int argc, char* argv[])
{
  // configure kernel with *.config files
  vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
  for(int i=1; i<argc; i++)
  {
    // loading config file passed on command line...
    kernel->loadConfigFile(argv[i]);
  }

  // start the kernel
  kernel->start();

  // set the application for the kernel to run
  MyApplication* application = new MyApplication();
  kernel->setApplication(application);

  // Block until the kernel exits.
  kernel->waitForKernelStop();

  return 0;
}
```
Index

A
application object, 26, 26, 26
  base interface of, 32
  benefits of, 28
  frame functions, 34
  initialization, 33
  overview, 26
application programming
  getting input, 59
  OpenGL, 63
    clearing color and depth buffers, 65
    context-specific data, 67
    drawing, 65
  OpenGL Performer, 73
    scene graph access, 75
    scene graph initialization, 75
  OpenSG, 78
    scene graph access, 80
    scene graph initialization, 79
  where to get device input, 60
applications
  application object overview, 26
  basics, 26
  device access, 56
  main function
    structure, 29
    use, 29
  starting, 29
  writing, 59

D
device aliases
  examples of, 57
device interfaces
  as smart pointers
    (see also smart pointer)
  initialization of, 57
device proxies
  access through device interfaces
    (see also gadget::DeviceInterface<T>)
Draw Manager, 26
  application classes, 35
  OpenGL, 35
  OpenGL Performer, 37

F
frame, 30
  definition of, 31
  frame of execution, 31

G
gadget::DeviceInterface
  instantiations of, 57
gadget::DeviceInterface<T>, 55
  description of, 56
  details, 57
gadget::Proxy, 58
  as pointer to physical device, 58
  description of, 58
  details, 58
  getData() method, 58
GLUT
  porting to VR Juggler, 89
gmtl::Matrix44f, 46
  adding, 50
  assigning, 49
  compared to C++ matrices, 46
  converting to pfMatrix, 54
  creating, 48
  description of, 46
  details, 55
  equality comparison, 49
  extracting transformation information, 54
  inverting, 50
  making
    Euler rotation, 53
    identity, 52
    scale transformation, 54
  translation transformation, 53
  multiplying, 51
Index

scaling, 52
subtracting, 51
transposing, 50
zeroing, 53
gmtl::Vec3f
adding, 45
assigning, 44
converting to pfVec3, 43
creating, 41
cross product, 45
details, 46
dividing by a scalar, 43
dot product, 44
equality comparison, 44
inverting, 42
length of, 42
multiplying by a scalar, 42
normalizing, 42
subtracting, 45
transforming by a matrix
full, 45
gmtl::Vec4f
adding, 45
assigning, 44
creating, 41
details, 46
dividing by a scalar, 43
dot product, 44
equality comparison, 44
inverting, 42
length of, 42
multiplying by a scalar, 42
normalizing, 42
subtracting, 45
transforming by a matrix
full, 45
gmtl::Vec<S, T>, 40
description of, 40

input device types, 59

main function, 26

OpenGL rendering contexts, 67

pfMatrix
converting from gmtl::Matrix44f
(see also gmtl::Matrix44f)

pfVec3
converting from gmtl::Vec3f
(see also gmtl::Vec3f)
porting applications from
CAVElib, 84

GLUT, 89
proxy
application-level access, 56
definition of, 56

S
smart pointer, 56

T
tutorial
drawing a cube using display lists, 71
drawing a cube with OpenGL, 66
getting input, 62
loading a model with OpenGL Performer, 75
loading a model with OpenSG, 80

V
virtual platform, 26
vrj::GlApp
bufferPreDraw() method, 65
contextInit() method, 69
contextPreDraw() method, 70
draw() method, 65
extensions to vrj::App, 64
vrj::Matrix
using with OpenGL, 55
vrj::OpenSGApp
extensions to vrj::GlApp, 79
getSceneRoot() method, 80
initScene() method, 79
vrj::PfApp
appChanFunc() method, 77
configPWin() method, 77
drawChan() method, 78
extensions to vrj::App, 74
getFrameBufferAttrs() method, 77
getScene() method, 75
initScene() method, 75
postDrawChan() method, 78
preDrawChan() method, 78
preForkInit() method, 77
Biography for Dr. Carolina Cruz-Neira

Dr. Carolina Cruz-Neira, one of the co-developers of the CAVE™ system and author of the CAVE Library™, is the Associate Director of the Virtual Reality Applications Center at Iowa State University, where she is Associate Professor in the Industrial and Manufacturing Systems Engineering Department, and leads research projects funded by such institutions as Deere & Company and Procter & Gamble. She is also the co-founder and vice president of Glass House Studio, LLC.

In 1997, Dr. Cruz was featured in BusinessWeek magazine as a “rising research star” in the new generation of computer science pioneers. In March of 2000, Dr. Cruz received the Iowa State Foundation Award for Early Achievement in Research. In June 2001, she received the Boeing A.D. Welliver Award, and spent the summer working at Boeing sites around the country. In April 2002, she was named an Eminent Engineer by the Tau Beta Pi Engineering Honors Society. She is also a Senior member of IEEE since 2001. Her research as a computer engineer has been driven by her goal to support applicability and simplicity. These two factors have defined her three current areas of work, all of them in the context of virtual reality (VR): 1) Complex software systems that integrate various hardware components to create advanced collaborative immersive environments; 2) Applications of VR technology in science, engineering, and art; 3) Usability studies of working virtual environments.

Dr. Cruz earned a Ph.D. in Electrical Engineering and Computer Science at the Electronic Visualization Laboratory at University of Illinois at Chicago in 1995. In 1991, she received a Master’s degree in EECS at the University of Illinois at Chicago and graduated Cum Laude in Systems Engineering at the Universidad Metropolitana in Caracas, Venezuela, in 1987.
Biography for Allen Bierbaum

Allen Bierbaum is a Ph.D. candidate at the Virtual Reality Applications Center (VRAC) at Iowa State University. His research interests include software engineering and tools for virtual reality, distributed VR environments, and real-time rendering. He is one of the developers of VR Juggler and through this work he has been involved in the development of numerous VR applications. His current Ph.D. research focuses on software architectures for large scale distributed virtual environments.

Mr. Bierbaum has produced many refereed publications, technology demonstrations, and invited presentations. These include presentations at SIGGRAPH, IEEE VR, SUPERCOMPUTING, Immersive Projection Technology (IPT) workshops, and CAVE Programming Workshops.
Biography for Patrick Hartling

Patrick Hartling is a graduate research assistant at the Virtual Reality Applications Center (VRAC) at Iowa State University. His research interests include collaboration between heterogeneous virtual reality systems, collaboration in large-scale, widely distributed environments, and interoperability between programming languages. He is one of the developers of VR Juggler and has been involved in the development of many VR middleware projects. He is currently pursuing his Ph.D. in Computer Engineering. He holds a B.S. and an M.S. in Computer Science. He was invited to teach courses as part of the 1st and 2nd Scandinavian CAVE Programming Workshops where he lectured about writing VR Juggler applications. In addition, he has spoken about his research and virtual reality in general at several international conferences and workshops. Mr. Hartling has done VR Juggler consulting work for several companies including Deere & Co., Intelligent Light, and Glass House Studio.