VR Juggler

The Programmer's Guide

VR Juggler: The Programmer's Guide

Version 2.2

Published $Date: 2007-12-13 22:21:26 -0600 (Thu, 13 Dec 2007) $

Table of Contents

List of Figures

List of Tables

List of Examples

Part I. Introduction

Table of Contents

Chapter 1. Getting Started

In this book, we present a “how-to” for writing VR Juggler applications. We will explain concepts used in VR Juggler and present carefully annotated example code whenever appropriate. There are two groups of people who should read this book:

1. Those who are required to read it in order to do a project for work or school. To those in this category, fear not—VR Juggler is very simple to use after getting through the initial learning stages. It is a powerful tool that will allow the creation of interesting and powerful applications very quickly.
2. Those who are just interested in creating compelling, interesting VR applications. VR Juggler facilitates the construction of extremely powerful applications that will run on nearly any combination of hardware architecture and software platform.

Necessary Experience

To help readers get the most from this book, recommendations follow to provide an idea of what previous experience is necessary. Various programming skills are needed, of course, but programming for VR requires more than just knowledge of a given programming language. VR Juggler takes advantage of many programming design patterns and advanced concepts to make it more powerful, more flexible, and more extensible. A good background in mathematics is helpful for performing the myriad transformations that must be applied to three-dimensional (3D) geometry.

Required Background

To get the most from this chapter, there are a few prerequisites:

- C++ programming experience
- Some graphics programming background (e.g., OpenGL, OpenGL Performer, etc.)
- Reasonable mathematical background (linear algebra knowledge is very useful)

For some of the advanced sections of this book, it is recommended that readers review the VR Juggler architecture book. This is optional, though it may be helpful in gaining a quicker understanding of some topics and concepts.

Other VR Software Tools

Readers who already have experience with other VR software development environments can easily skim through this book and find the relevant new information. The book is designed for easy skimming. Simply look at the headings to get a good determination of what should be read and what may be skipped.

Organization

This book is organized into three main parts:

1. Introduction: The introduction to the key VR Juggler application development concept, application

objects. We also describe common helper classes that simplify the process of writing applications.

2. Writing applications: The presentation of application development including how to get input from devices and how to write applications for each of the supported graphics application programmer interfaces (APIs).

3. Advanced topics: An extension of the previous chapters showing how to incorporate run-time reconfiguration into applications and how to write multi-threaded applications.

Chapter 2. 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.

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*.

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 set of pre-defined interfaces. The VR Juggler kernel controls the application's processing time by calling the object's interface implementation methods.

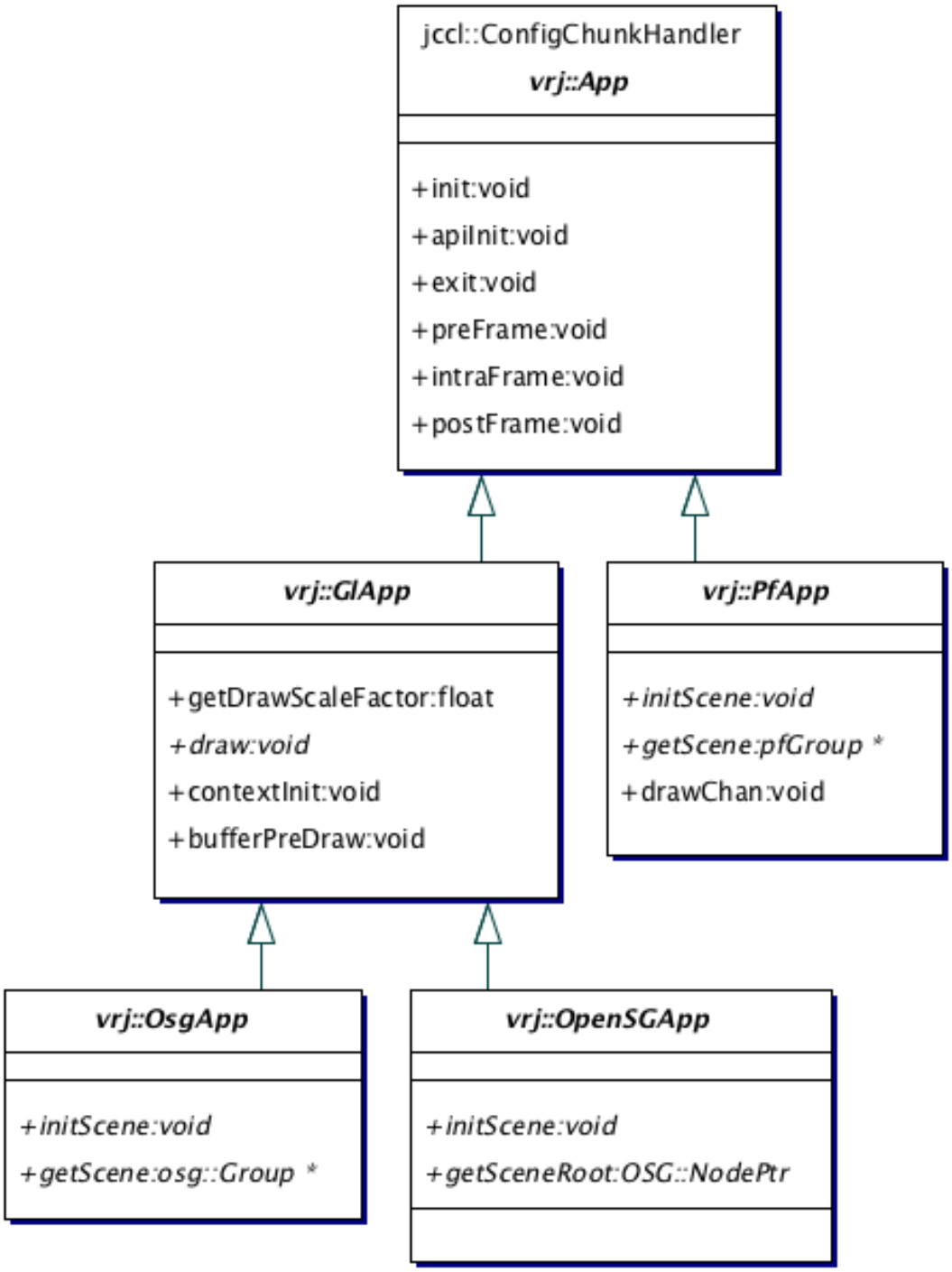
In traditional programs, the `main()` function defines the point where the *thread of control* enters the application. After the `main()` function is called, the application starts performing any necessary processing. When the operating system (OS) starts the program, it gives the `main()` function some 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.

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 2.1, “`vrj::App` hierarchy” 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 2.1. `vrj::App` hierarchy



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.

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 CAVElibs™ 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.

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.

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 possibly 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.

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.

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.

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.

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.

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.

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

   int main(int argc, char* argv[])
 5 {
1    vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
2    simpleApp* app      = new simpleApp();         // Create the app object

3    kernel->loadConfigFile(...);             // Configure the kernel
4    kernel->start();                         // Start the kernel thread
5    kernel->setApplication(app);             // Give application to kernel
     kernel->waitForKernelStop();             // Block until kernel stops

     return 0;
15 }
```

1. This line finds (and may create) the VR Juggler kernel. The kernel reference is stored in the handle `kernel` that we can use it later.
2. Here 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.

5 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.

Mac OS X Considerations

VR Juggler 2.2 introduces support for Cocoa on Mac OS X. Cocoa itself is quite different than the X11 or Win32 APIs as far as restrictions on threading and the implementation of the `main()` function. The fundamental issue, however, is that both the `NSApplication` singleton object and the `vrj::Kernel` singleton object want to be in charge of application execution. A balance has been struck that generally allows VR Juggler applications to look and act the same on Mac OS X as on any other platform while still taking advantage of unique features that Cocoa and Mac OS X have to offer.

Application Bundles

First and foremost, when using a Cocoa-aware version of VR Juggler, applications have to be constructed as bundles. The details of OS X application bundles are far beyond the scope of this document. However, VR Juggler provides just about everything that is required to make an application bundle. This makes sense because VR Juggler is an application framework that dictates how applications are written and executed. Thus, it also provides the core information required for proper application construction and execution on Mac OS X.

To get started, it is helpful to understand what the application bundle will look like when we have everything in place. An application bundle is a directory structure with a special name. For example, `MPApp.app` will show up in the Finder as an application named “MPApp” that can be double-clicked. The name can contain spaces if so desired.

Under the base directory is the `Contents` subdirectory. It will contain the files `Info.plist` and `PkgInfo`.

Next, there are two subdirectories `MacOS` and `Resources`. Compiled binaries should normally go into the `MacOS` subdirectory. If nothing else, the bundle executable (the compiled application binary) *must* go in the `MacOS` directory. Shared libraries can go into the `Resources` directory if desired. Generally, though, the `Resources` directory will contain platform-independent data files.

In the resources directory, there will be another subdirectory `en.lproj`[1] which in turn contains `MainMenu.nib` from `$VJ_BASE_DIR/share/vrjuggler/data/bundle`. This is a critical part of the application bundle, and it is very important that this NIB behave correctly. The NIB defines the user interface for the VR Juggler application, and what is provided with VR Juggler is set up and ready to go for the most common cases. It is possible to use a different `MainMenu.nib`, but customizing it will require care.

> The ability to customize the user interface of the VR Juggler application by using a different `MainMenu.nib` and a different application delegate (see the section called “Making a Custom `NSApplication` Delegate on Mac OS X”) makes VR Juggler on Mac OS X unique with respect to other platforms. Whereas the user interface used with Microsoft Windows® and the X Window System is essentially hard coded, the Mac OS X Cocoa interface has been designed with flexibility in mind. Simply put, the Cocoa support in VR Juggler leverages the dynamic nature of Cocoa to provide features that are not so easily available on other platforms.

[1] This is the English language project data. Translations of `MainMenu.nib` would go into the appropriate language-specific subdirectory.

Info.plist

The files and directories needed for application bundle creation can be found in the directory $VJ_BASE_DIR/share/vrjuggler/data/bundle. The first of these is Contents/Info.plist, the basic property list for an application bundle. Open it with the Property List Editor application and change the @APP_NAME@ strings accordingly for the name of the application being constructed. For the CFBundleExecutable property, be sure to change @APP_NAME@ to be the name of the executable that will be in the Contents/MacOS directory. Other properties to change are those that include copyright and version information.

PkgInfo

The contents of Contents/PkgInfo define the application bundle as an application bundle. The contents of the file will often be APPL????, though other characters are allowed in place of the ???? part.

Resources Directory

The Contents/Resources directory contains data files related to application execution. The file vrjuggler.plist, the use of which is highly recommended, must be put in this directory. The application bundle icon file (a file with the extension .icns) is also stored in this directory. A default icon file, vrjuggler.icns, can be used, or a custom one can be made. The file to use must be named in Contents/Info.plist.

vrjuggler.plist

For VR Juggler application bundles, a useful file is vrjuggler.plist. The default version of this file from $VJ_BASE_DIR/share/vrjuggler/data/bundle disables VR Juggler configuration file loading through NSApplication channels and identifies the NSApplication delegate class type that will be used (see the section called “Making a Custom NSApplication Delegate on Mac OS X”). These are set using the properties VRJConfigHandling and VRJDelegateClass respectively.

The “configuration file loading through NSApplication channels” bit has to do with associating VR Juggler configuration files with application bundles. There are different means by which data can be delivered to applications on Mac OS X. For example, double-clicking on a file in the Finder (or selecting Open With in the context menu for the file) causes a registered handler application to be opened. The file is given to the application once it has opened through the NSApplication event system. By setting the VRJConfigHandling property to false in Contents/Resources/vrjuggler.plist, this capability is disabled for the application bundle in question. That is, the application bundle will ignore the NSApplication events pertaining to configuration files to be loaded as a result of double-clicking on the said files. Configuration files can still be opened on the fly using the File menu defined in the default NIB (see below).

en.lproj Directory

The en.lproj subdirectory of Contents/Resources contains information translated in the English language. The most important item in this directory is MainMenu.nib (discussed next), but the optional file $VJ_BASE_DIR/share/vrjuggler/data/bundle/InfoPlist.strings, or some other version of same, can be copied into this directory. Other language-specific directories can be created as subdirectories of Contents/Resources as necessary. They, too, must contain a MainMenu.nib.

MainMenu.nib

This is the NIB for the VR juggler application bundle. It has been designed using Interface Builder with a simple user interface that knows how to interact with a VR Juggler application. The interface includes the File menu with the item for opening VR Juggler configuration files on the fly. Other menus and

menu items can be added by making a custom MainMenu.nib. Using the version from `$VJ_BASE_DIR/share/vrjuggler/data/bundle` as a starting point is a good idea. Note that `MainMenu.nib` is a directory, and it should be copied recursively to `Contents/Resources/en.lproj`.

Application Execution

The `main()` function implementation show in the section called "Structure of a `main()` Function" will work without any problems on Mac OS X. However, many of the VR Juggler sample and test applications have a slightly more complicated `main()` function. In particular, these applications usually determine whether the user has passed in any arguments through the command line and exit with an error message explaining how to run the application if none were given. The example below shows how this is commonly done with an `if` statement before anything else:

```
 1 @include <iostream>
   #include <cstdlib>
   #include <vrj/Kernel/Kernel.h>
   #include <simpleApp.h>
 5
   int main(int argc, char* argv[])
   {
      if ( argc <= 1 )
      {
10       std::cout << "Usage: " << argv[0]
                   << "cfgfile[0] cfgfile[1] ... cfgfile[n]"
                   << std::endl;
         std::exit(EXIT_FAILURE);
      }
15
      vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
      simpleApp* app      = new simpleApp();         // Create the app object

      kernel->loadConfigFile(...);             // Configure the kernel
20    kernel->start();                         // Start the kernel thread
      kernel->setApplication(app);             // Give application to kernel
      kernel->waitForKernelStop();             // Block until kernel stops

      return 0;
25 }
```

This approach will not necessarily work on Mac OS X because users normally expect to be able to launch an application by double-clicking on its icon in the Finder and then loading data into it through a GUI. If a user launched the above application that way, the application would exit immediately, and the user would have to open the Console application to find out what went wrong. Launching from the command line, while perfectly valid on Mac OS X, is simply not what users expect. At the same time, remote launching of VR juggler applications in a cluster will almost certainly require launching without the use of the Finder and passing in arguments through `argv`. This is definitely the case when using Maestro [https://realityforge.vrsource.org/trac/maestro/].

If a VR Juggler application will be run on Mac OS X, the programmer has to decide whether command line processing is critical to the execution of the application. There are three options available to VR Juggler programmers on Mac OS X: rely exclusively on command line arguments to launch, handle no command line arguments, or allow the use of command line arguments but do not require them. Given that VR Juggler applications written on all other platforms are highly likely to rely exclusively on command line arguments to launch, the first choice is expected to be the most commonly chosen approach. Nevertheless, we will explain all three.

Relying Exclusively on Command Line Arguments to Launch

If a VR Juggler application used on Mac OS X will rely solely on command line arguments to launch, it is operating in exactly the same manner as on all other platforms. The `main()` function can be written as is shown in this document to exit if no command line arguments are supplied. There is still one more thing to do, though. In the `Resources` directory of the application bundle, there must be a property list file in the bundle called `Contents/Resources/vrjuggler.plist`. This property list must have the property `VRJConfigHandling` set to `false`. This is the default setting for this property if the `vrjuggler.plist` file that comes with VR Juggler was used in constructing the application bundle.

To execute the application, run the program in `<appname>.app/Contents/MacOS` from a command prompt and pass in the command line arguments. The appropriate event handling will be set up so that the application will behave just as any other Mac OS X application.

Handling No Command Line Arguments at Launch Time

If no command line arguments are to be handled (i.e., the application is to behave the same as typical Mac OS X applications), then the `main()` function shown above needs to change. Specifically, it cannot exit with an error message if no command line arguments are passed in. To keep the application portable, a preprocessor conditional can be used, as shown in Example 2.1, “Ignoring Command Line Arguments on Mac OS X”. Then, in `Contents/Resources/vrjuggler.plist`, set the `VRJConfigHandling` property to `true`. To execute the application, double-click the application bundle icon in the Finder or use the **open** command from a command prompt.

Example 2.1. Ignoring Command Line Arguments on Mac OS X

```
 1 @include <iostream>
   #include <cstdlib>
   #include <vrj/Kernel/Kernel.h>
   #include <simpleApp.h>
 5
   int main(int argc, char* argv[])
   {
   #if ! defined(VRJ_USE_COCOA)
      if ( argc <= 1 )
10    {
         std::cout << "Usage: " << argv[0]
                   << "cfgfile[0] cfgfile[1] ... cfgfile[n]"
                   << std::endl;
         std::exit(EXIT_FAILURE);
15    }
   #endif

      vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
      simpleApp* app      = new simpleApp();         // Create the app object
20
   #if ! defined(VRJ_USE_COCOA)
      kernel->loadConfigFile(...);            // Configure the kernel
   #endif
      kernel->start();                        // Start the kernel thread
25    kernel->setApplication(app);            // Give application to kernel
      kernel->waitForKernelStop();            // Block until kernel stops

      return 0;
   }
```

Allowing Optional Use of Command Line Arguments

A compromise can be struck between the previous two options by allowing optional use of command

line arguments. The compromise is simple: do not require command line arguments but still handle them if they are given. The form of the main() function that allows this is shown in Example 2.2, “Handling Optional Command Line Arguments on Mac OS X”. Note that the `#if` around the call to `vrj::Kernel::loadConfigFile()` has been removed.

Example 2.2. Handling Optional Command Line Arguments on Mac OS X

```
 1 @include <iostream>
   #include <cstdlib>
   #include <vrj/Kernel/Kernel.h>
   #include <simpleApp.h>
 5
   int main(int argc, char* argv[])
   {
   #if ! defined(VRJ_USE_COCOA)
      if ( argc <= 1 )
10    {
         std::cout << "Usage: " << argv[0]
                   << "cfgfile[0] cfgfile[1] ... cfgfile[n]"
                   << std::endl;
         std::exit(EXIT_FAILURE);
15    }
   #endif

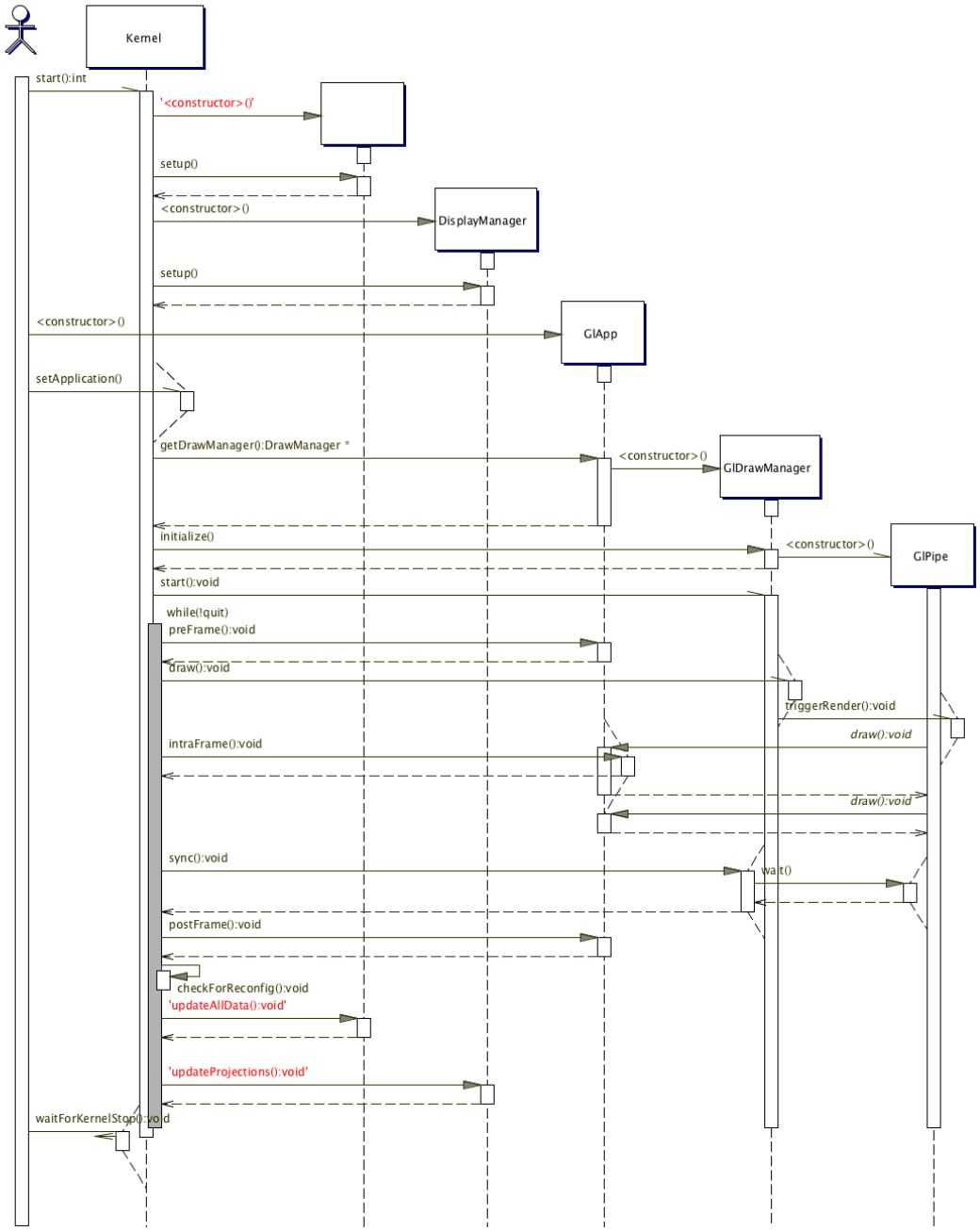
      vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
      simpleApp* app      = new simpleApp();         // Create the app object
20
      kernel->loadConfigFile(...);             // Configure the kernel
      kernel->start();                         // Start the kernel thread
      kernel->setApplication(app);             // Give application to kernel
      kernel->waitForKernelStop();             // Block until kernel stops
25
      return 0;
   }
```

The only decision left to make is what value to use for the `VRJConfigHandling` property in `Contents/Resources/vrjuggler.plist`. If we set it to `false` (recall that that is the default in the `$VJ_BASE_DIR/share/vrjuggler/data/bundle/vrjuggler.plist` version), the application cannot be used as a handler for VR Juggler configuration files. If we set it to true, we have to be prepared for configuration files to come in other than through the command line or through the use of the File menu. The default `NSApplication` delegate (`VRJBasicDelegate`) can deal with either case. The default behavior for VR Juggler sample applications is to allow configuration files to be specified on the command line through the use of a `main()` function similar to that shown in Example 2.2, “Handling Optional Command Line Arguments on Mac OS X” and to leave the property `VRJConfigHandling` in `Contents/Resources/vrjuggler.plist` set to `false`.

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 2.2, “Kernel loop sequence”. 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 2.2. Kernel loop sequence



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 2.2, “Kernel loop sequence”; 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.

Base Application Object Interface

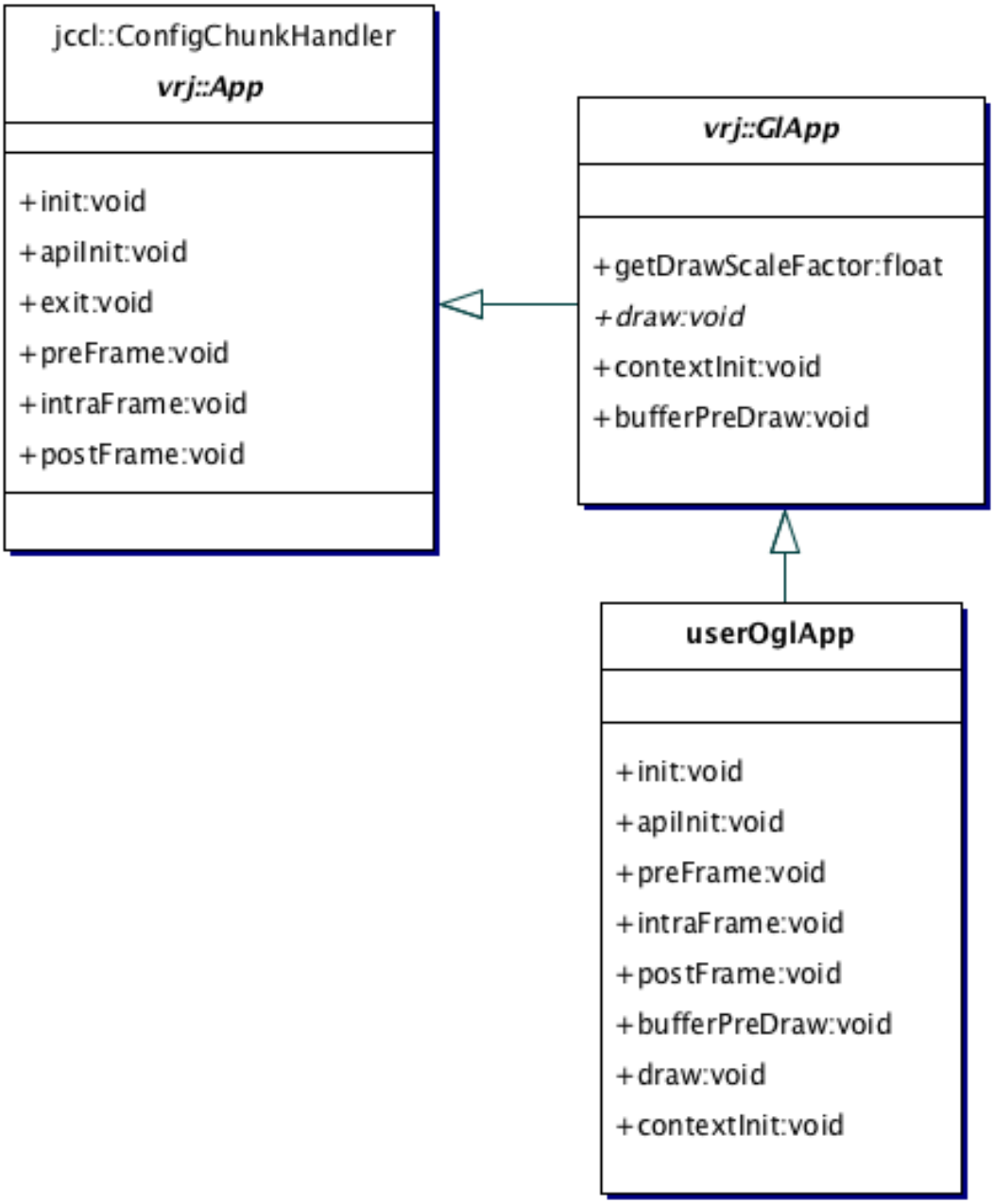
Within this section, we provide a brief overview of the member functions from the base VR Juggler application interface. This interface is defined by `vrj::App`, and the member functions are shown in Figure 2.3, “Application object interface”. Refer to Figure 2.2, “Kernel loop sequence” for a visual 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 2.3. Application object interface



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.

`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.

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.

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.

`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.

Timing

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

Uses

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

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.

`vrj::App::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:

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

Overriding this method means changing the rendering scale factor used by the VR Juggler Draw Managers. 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.

`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.

Timing

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

Uses

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

`vrj::App::latePreFrame()`

The `latePreFrame()` method is called after `preFrame()` and after shared application-specific data is synchronized among the cluster nodes (see the section called “Cluster Application Programming” for more details) but before the scene is rendered. Scene graph-based application objects making use of application-specific data in a cluster configuration should perform scene graph updates based on the most recently received copy of the shared application data. Application objects not using a scene graph can make state updates in this method or in the rendering method (`draw()` in the case of `vrj::GlApp`). The writer node must have written to the shared application data in `preFrame()` to minimize the latency of the data.

Timing

This method is called after application-specific data is sychronized among the cluster nodes but before triggering rendering of the current frame.

Uses

When using shared application-specific data with a scene-graph based application object in a cluster configuration, the nodes that read from the shared data (those where `cluster::UserData<T>::isLocal()` returns false) should perform state updates based on the freshly received update to the shared data.

`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 be-

cause 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.

Timing

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

Uses

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

`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).

Timing

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

Uses

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

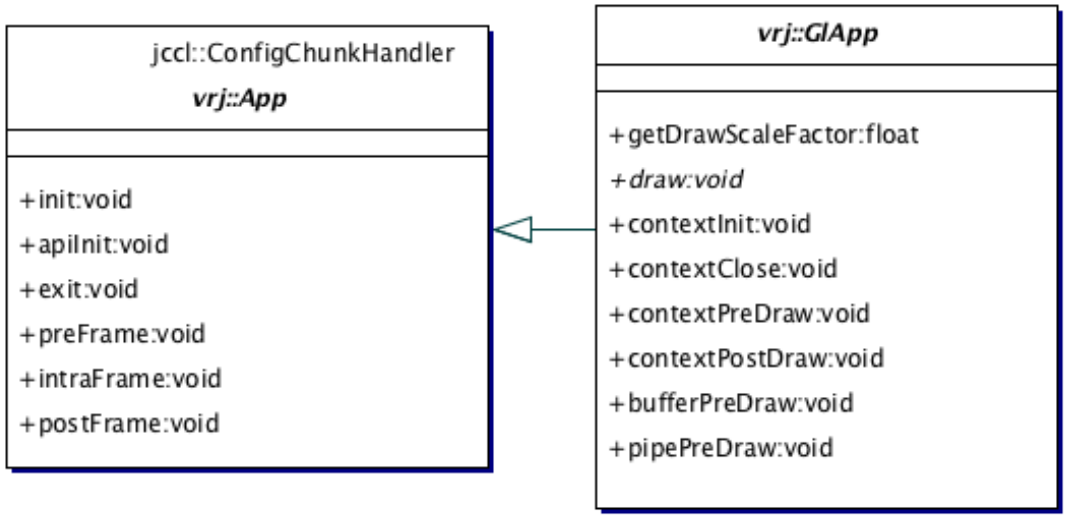
Draw Manager-Specific Application Classes

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.

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 2.4, “`vrj::GlApp` interface extensions to `vrj::App`”. 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 the section called “OpenGL Applications”, found in Chapter 4, *Application Authoring Basics*.

Figure 2.4. `vrj::GlApp` interface extensions to `vrj::App`



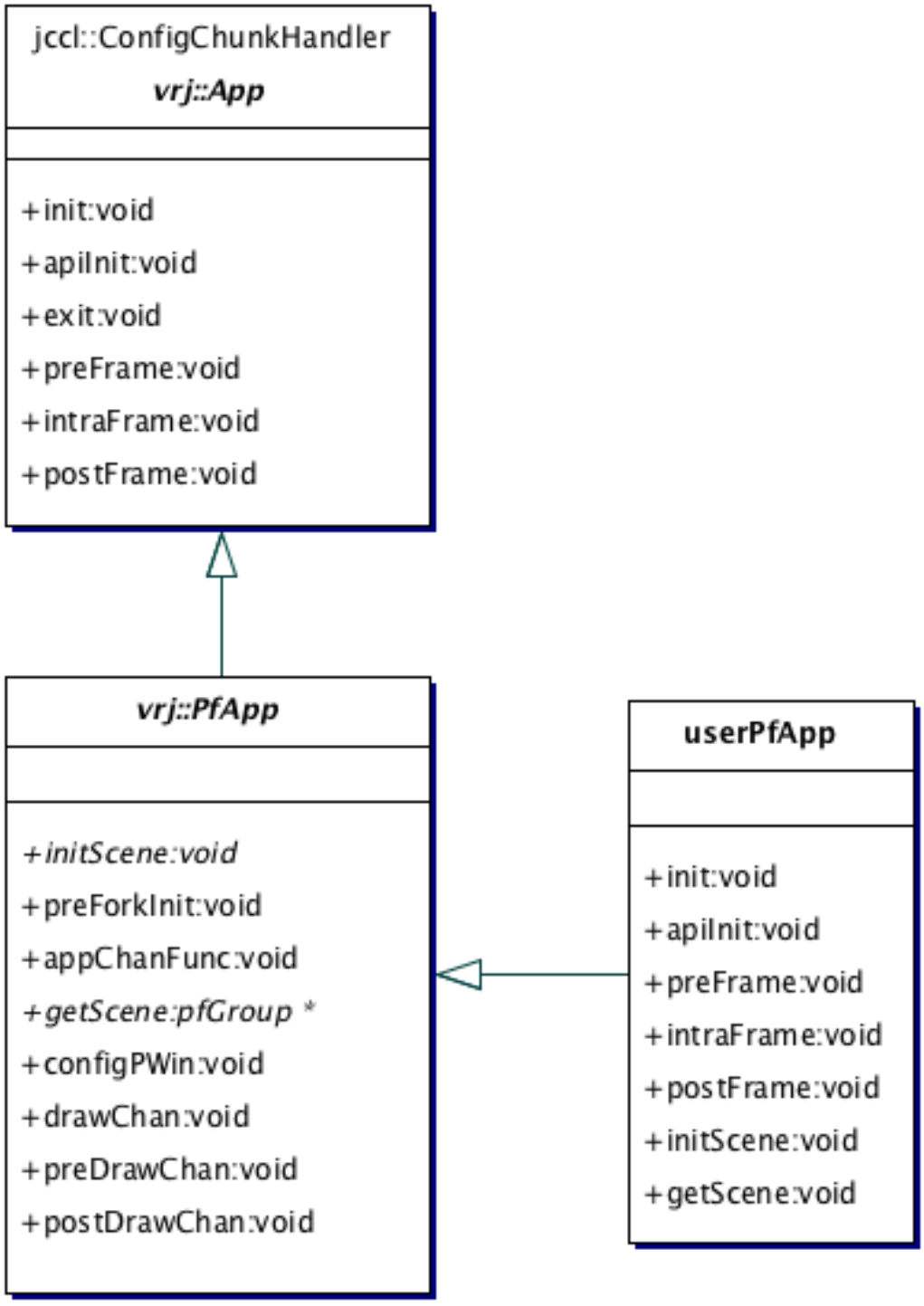
`vrj::GlApp::draw()`

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.

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 2.5, “`vrj::PfApp` interface extensions to `vrj::App`”. The following is a description of only two methods in the `vrj::PfApp` interface. More detailed discussion on this class is provided in the section called “OpenGL Performer Applications”, found in Chapter 4, *Application Authoring Basics*.

Figure 2.5. `vrj::PfApp` interface extensions to `vrj::App`



`vrj::PfApp::initScene()`

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

`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 3. 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.

The `gmtl::Vec<S, T>` Helper Class

This section is intended to provide an introduction to how the helper class `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.

High-Level Description

The class `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 `gmtl::Vec3f` object can be thought of as a vector of the form <x, y, z>. Similarly, a `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.

`gmtl::Vec3f` and `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.

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.

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>:

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

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

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

or individually:

```
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:

```
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).

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:

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

vec2 = -vec1;
```

The vector `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.)

Normalization

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

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

gmtl::normalize( vec1 );
```

The vector `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 `vec` has already been declared before this point):

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

Length Calculation

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

```
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.

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:

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

vec2 = 3 * vec1;
```

(The order of the factors in the multiplication can be swapped depending on preference or need.) Here, `vec2` gets the value <3.0, 4.5, -3.0>.

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

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

vec1 *= 3;
```

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

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:

```
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:

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

vec1 /= 3;
```

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

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:

```
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:

```
pfVec3 pf_vec;
gmtl::Vec3f vj_vec;

// Do stuff to pf_vec...

vj_vec = vrj::GetVjVec(pf_vec);
```

Here again, `pf_vec` is passed by reference for efficiency, and `vj_vec` gets a copy of a `gmtl::Vec3f`. Both of these functions are found in the header `vrj/Draw/Pf/PfUtil.h`.

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:

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

vec2 = vec1;
```

After the assignment, `vec2` has the value <-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.

Equality/Inequality Comparison

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

```
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

```
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

```
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.

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:

```
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.

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`.

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>.

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>.

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 `gmtl::Matrix44f` transformation matrix `xform_mat`:

```
gmtl::Vec3f vec(1.0, 1.0, 1.0), result_vec;

gmtl::xform(result_vec, xform_mat, vec1);
```

Depending on the transformations contained within `xform_mat`, `result_vec` will be transformed fully. The operation as a mathematical equation would be:

$$V' = M * V$$

where V and V' are vectors and M is a 4×4 transformation matrix.

The Gory Details

The details behind `gmtl::Vec3f` and `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 `getData()`. For example, this access can be used in the following way:

```
gmtl::Vec3f pos(4.0, 1.0982, 10.1241);

glVertex3fv(pos.getData());
```

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

In general, the `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 `getData()` follows:

```
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;
```

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.

The `gmtl::Matrix44f` Helper Class

This section is intended to provide an introduction into how the helper class `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 `gmtl::Matrix44f`.

High-Level Description

Abstractly, `gmtl::Matrix44f` represents a 4×4 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 4×4 C++ array of floats, but there is one crucial difference: a `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++ multidimensional 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 performed 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.

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.

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:

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

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

```
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:

```
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:

```
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:

```
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, `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, `val` is assigned the value 3.8. Both of these operations are row-major.

Assignment

Assigning one `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;

mat2 = mat1;
```

This makes a *copy* of `mat1` in `mat2` which can be a slow operation.

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

```
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.

Transposing

The transpose operation works conceptually as $matrix_1 = transpose\left(matrix_2\right)$. The code is then:

```
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.

Finding the Inverse

The inverse operation works conceptually as $matrix_1 = inverse\left(matrix_2\right)$. The code is then:

```
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.

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.

```
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::Matrix44f mat3;
```

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

```
gmtl::add(mat3, mat1, mat2);
```

Using the overloaded + operator:

```
mat3 = mat1 + mat2;
```

The result is stored (via a copy) in `mat3`.

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 `gmtl::Matrix44f`: using the `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 `gmtl::Matrix44f` objects.

```
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::Matrix44f mat3;
```

Using the `gmtl::sub()` method:

```
gmtl::sub(mat3, mat1, mat2);
```

Using the overloaded - operator:

```
mat3 = mat1 - mat2;
```

The result is stored (via a copy) in `mat3`.

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 `gmtl::Matrix44f`: using the `gmtl::mult()`

function or using the overloaded * operator. We recommend the use of the `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 `gmtl::Matrix44f` objects.

```
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::Matrix44f mat3;
```

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

```
gmtl::mult(mat3, mat1, mat2);
```

Using the overloaded * operator:

```
mat3 = mat1 * mat2;
```

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:

```
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 $mat_1 = mat_1 * mat_2$ so that the second matrix (`mat2`) comes as the second factor. The same result can be achieved using the overloaded *= operator:

```
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:

```
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 $mat_1 = mat_2 * mat_1$ so that the second matrix (`mat2`) comes as the first factor. In both cases, the result of the multiplication is stored in `mat1`.

Scaling by a Scalar Value

Scaling the values of a matrix by a scalar value can be done using two different methods: the `setS-`

cale() method or the overloaded * and / operators that take a single scalar value and returns a 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 gmtl::setScale() function works as:

```
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);
```

Making an Identity Matrix Quickly

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

```
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::identity(mat1);
```

Of course, simply declaring mat1 with no arguments would achieve the same result, but that is not such an interesting example.

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:

```
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.

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:

```
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.

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):

```
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:

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

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

Making a Scale Transformation Matrix

To make a transformation matrix that only scales, a simple method is provided. It works as follows:

```
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.

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:

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

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

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

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

```
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`.

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:

```
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:

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

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 3.1, “Row-major access indices”

Table 3.1. Row-major access indices

(0,0)	(0,1)	(0,2)	(0,3)	<--- Array
(1,0)	(1,1)	(1,2)	(1,3)	<--- Array
(2,0)	(2,1)	(2,2)	(2,3)	<--- Array
(3,0)	(3,1)	(3,2)	(3,3)	<--- Array

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 3.2, “Column-major access indices”.

Table 3.2. Column-major access indices

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)
^	^	^	^
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.

Device Proxies and Device Interfaces

When writing a VR Juggler application object, direct access to hardware devices is not allowed. This is a design choice that helps facilitate the portability of VR applications by preventing them from depending on specific input device makes and models. 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.

Proxies are acquired through the Gadgeteer Input Manager, but the process is not entirely straightforward. To assist with the use of device proxies, Gadgeteer provides what are called *device interfaces*. Device interfaces hide the details of proxy acquisition through the Input Manager, but the concept of device interfaces is something that often causes confusion for those new to VR Juggler application programming. Two object-oriented design patterns are combined by device interfaces: smart pointers and proxies. Within this section, we aim to explain Gadgeteer device proxies and device interfaces clearly and simply. We begin with high-level descriptions of both and then move on to their use in VR Juggler application objects.

High-Level Description of Device Proxies

As noted above, access to input device data is granted through device proxies allocated by the Gadgeteer Input Manager. For each type of input device (digital, analog, keyboard/mouse, etc.), there is a device proxy class. 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 application object programmers will probably never need to know more about the interface of a type-specific device proxy than the return type of its data request method (usually named `getData()`). We will see more about the data request method in the explanation of device interfaces below. Most of the perceived complexity in the type-specific proxy classes is only important to Gadgeteer's internal maintenance of the active proxies. The following is the complete list of proxy classes in Gadgeteer 1.0 (used by VR Juggler 2.0):

`gadget::AnalogProxy`	The proxy type for analog input devices (those of type `gadget::Analog`). It is defined in the header file `gadget/Type/AnalogProxy.h`. The return type of `gadget::AnalogProxy::getData()` is float.
`gadget::CommandProxy`	The proxy type for command-oriented input devices (those of type `gadget::Command`). It is defined in the header file `gadget/Type/CommandProxy.h`. The return type of `gadget::CommandProxy::getData()` is int.
`gadget::DigitalProxy`	The proxy type for digital (on/off) input devices (those of type `gadget::Digital`). It is defined in the header file `gadget/Type/DigitalProxy.h`. The return type of `gad-`

	`get::DigitalProxy::getData()` is gadget::Digital::State, an enumerated type. This means that values returned by `gadget::DigitalProxy::getData()` can be treated as integers.
`gadget::GloveProxy`	The proxy type for glove input devices (those of type `gadget::Glove`). It is defined in the header file `gadget/Type/GloveProxy.h`. The return type of `gadget::GloveProxy::getData()` is `gadget::GloveData`.
`gadget::KeyboardMouseProxy`	The proxy type for keyboard/mouse input handlers (those of type `gadget::KeyboardMouse`). It is defined in the header file `gadget/Type/KeyboardMouseProxy.h`. There is no method `gadget::KeyboardMouse::getData()`. Rather, the method to use for querying input data is `gadget::KeyboardMouseProxy::getEventQueue()`, the return type of which is `gadget::KeyboardMouse::EventQueue`. We explain more about this below in the section called "Using `gadget::KeyboardMouseInterface`".
`gadget::PositionProxy`	The proxy type for position tracking input devices (those of type `gadget::Position`). It is defined in the header file `gadget/Type/PositionProxy.h`. The return type of `gadget::PositionProxy::getData()` is `gmtl::Matrix44f`. The method `gadget::PositionProxy::getData()` takes an optional float paramter that indicates the units to use for the returned transformation matrix. The value must be a conversion factor from meters to the desired units.
`gadget::StringProxy`	The proxy type for string (text- or word-driven) input devices (those of type `gadget::String`). It is defined in the header file `gadget/Type/StringProxy.h`. The return type of `gadget::StringProxy::getData()` is `std::string`.

In summary, the important thing to know is that a proxy is a pointer to a physical device. Application object programmers should normally use the higher level device interface as the mechanism to acquire a proxy and read data from the proxied 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.

High-Level Description of Device Interfaces

Device interfaces are designed to act as wrappers around type-specific device properties. This is implemented through the (template) class `gadget::DeviceInterface<T>`. Applications could use the proxy classes directly, but as we have already noted, acquiring the desired proxies from the Gadgeteer Input Manager is not wholly straightforward. The type-specific instances of `gadget::DeviceInterface<T>` (such as `gadget::PositionInterface`, `gadget::DigitalInterface`, etc.) simplify acquisition of proxies. Thus, typical VR Juggler application objects will have one or more device interface member variables and no proxy member variables.

The class `gadget::DeviceInterface<T>` is a templated class based on the proxy type it wraps. The following is the complete list of available `gadget::DeviceInterface<T>` instantiations in

Gadgeteer 1.0 (used by VR Juggler 2.0):

`gadget::AnalogInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::AnalogProxy>`. It wraps the proxy type `gadget::AnalogProxy` and is used for reading data from analog devices. Include the header file `gadget/Type/AnalogInterface.h`.
`gadget::CommandInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::CommandProxy>`. It wraps the proxy type `gadget::CommandProxy` and is used for reading data from command-driven devices. Include the header file `gadget/Type/CommandInterface.h`.
`gadget::DigitalInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::DigitalProxy>`. It wraps the proxy type `gadget::DigitalProxy` and is used for reading data from digital (on/off) devices. Include the header file `gadget/Type/DigitalInterface.h`.
`gadget::GloveInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::GloveProxy>`. It wraps the proxy type `gadget::GloveProxy` and is used for reading data from glove devices. Include the header file `gadget/Type/GloveInterface.h`.
`gadget::KeyboardMouseInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::KeyboardMouseProxy>`. It wraps the proxy type `gadget::KeyboardMouseProxy` and is used for reading data from analog devices. Include the header file `gadget/Type/KeyboardMouseInterface.h`.
`gadget::PositionInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::PositionProxy>`. It wraps the proxy type `gadget::PositionProxy` and is used for reading data from position tracking devices. Include the header file `gadget/Type/PositionInterface.h`.
`gadget::StringInterface`	This is a typedef for the template instantiation `gadget::DeviceInterface<gadget::StringProxy>`. It wraps the proxy type `gadget::StringProxy` and is used for reading data from string (text- or word-driven) devices. Include the header file `gadget/Type/StringInterface.h`.

The typedefs are provided to make the application object 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 deference operator allows some extra work to be done 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. In user code, there will be instances of types such as `gadget::DigitalInterface`, `gadget::PositionInterface`, `gadget::KeyboardMouseInterface`, and the like. 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 that they encapsulate.

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::DigialInterface` gives exactly that information, and it quietly hides the messiness of dealing with the mouse, its driver, and its communication protocol.

Using Device Interfaces

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 override of the method `vrj::App::init()` method. Because we are dealing with a templated type (`gadget::DeviceInterface<T>`), every type-specific instantiation has the same interface. Hence, all type-specific device interfaces are initialized using the `gadget::DeviceInterface<T>::init()` method. This method takes a single string argument naming the proxy to which the interface will connect. The name can be the name of a proxy or a proxy alias, both of which are defined in VR Juggler configuration files. Example names are “VJHead”, “Wand”, “VJButton0”, and “Accelerate Button”. Using meaningful symbolic names 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:

```
gadget::PositionInterface head;

head.init("VJHead");
```

Remember that this has 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. 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 acquire data from the device proxy through the device interface 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 (in feet):

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

Believe it or not, the code really is that easy. Simply use the overloaded dereference operator to get access to the position proxy object hidden in `gadget::PositionInterface` to read data from the proxy. We now move on to explain the use of type-specific device interfaces.

Using `gadget::AnalogInterface`

Analog devices return floating-point data. As noted above, the return type of `gadget::AnalogProxy::getData()` is float. Behind the scenes, analog devices in Gadgeteer scale

their input so that application objects always receive it in the range 0.0 to 1.0 inclusive (also denoted [0.0,1.0] in mathematically oriented descriptions). Hence, application objects can always expect analog data to be in that range regardless of the specific type of analog device being used.

Using `gadget::CommandInterface`

Command-oriented devices were introduced in Gadgeteer 1.0 Beta 1. They are an evolving device type geared towards complex input that can be interpreted at an abstract level. In Gadgeteer 1.0, such input comes in the form of spoken phrases that are reinterpreted as commands identified by unique integer values. In future versions of Gadgeteer, this device type will be used for scalable gesture recognition. The return type of `gadget::CommandProxy::getData()` is int, and it is up to the person configuring the command-oriented device to set up the command-to-integer mappings.

Using `gadget::DigitalInterface`

Digital devices are those that have distinct on and off states. The method `gadget::DigitalProxy::getData()` returns the current state of such a device as a value of the enumerated type gadget::Digital::State. This type is defined to allow for easy on/off testing, but it also provides state toggling information. The possible values of gadget::Digital::State are `OFF` (integer value 0), `ON` (integer value 1), `TOGGLE_ON` (integer value 2), `TOGGLE_OFF` (integer value 3). In Example 3.1, “Using `gadget::DigitalInterface` in an Application Object”, we see some example uses of the information returned by `gadget::DigitalProxy::getData()`.

Example 3.1. Using `gadget::DigitalInterface` in an Application Object

```
 1 void MyApp::preFrame()
   {
      if ( mButton0->getData() )
      {
 5       // Set state for when mButton0 is pressed, has been pressed
         // since the last frame, or has been released since the
         // last frame ...
      }
      else
10    {
         // Set state for when mButton0 is not pressed ...
      }

      switch (mButton1->getData()
15    {
         case gadget::Digital::OFF:
            // Set state for when mButton1 is not pressed ...
            break;
         case gadget::Digital::ON:
20          // Set state for when mButton1 is pressed ...
            break;
         case gadget::Digital::TOGGLE_ON:
            // Set state for when mButton1 has been pressed since
            // the last frame ...
25          break;
         case gadget::Digital::TOGGLE_OFF:
            // Set state for when mButton1 has been released since
            // the last frame ...
            break;
30    }

      if ( mButton2->getData() == gadget::Digital::TOGGLE_ON )
      {
         // Set state when mButton2 goes "high" (is toggled on) ...
35    }
```

```
      else if ( mButton2->getData() == gadget::Digital::TOGGLE_OFF )
      {
         // Set state when mButton2 goes "low" (is toggled off) ...
      }
40 }
```

Using `gadget::GloveInterface`

Using `gadget::KeyboardMouseInterface`

Input read from a keyboard and a mouse is provided through `gadget::KeyboardMouseProxy`, instances of which are acquired through `gadget::KeyboardMouseInterface`. Unlike most other device proxy types, `gadget::KeyboardMouseProxy` does not have a `getData()` method. Rather, it has a method called `getEventQueue()` with return type `gadget::KeyboardMouse::EventQueue` that is the "event queue." Keyboard and mouse input is handled as events, either key press events or mouse events. Key press events come from the keyboard and are for both the pressing and releasing of individual keys or keys with modifiers (**CTRL**, **ALT**, and **SHIFT**). Mouse events include both the motion of the mouse in the X & Y axes and the pressing and releasing of mouse buttons which may or may not be associated with a keyboard modifier.

The event queue contains all the key press and mouse events that occurred since the last frame. Each event is contained in an object of type `gadget::EventPtr`[1]. The type `gadget::EventPtr` is a reference-counted smart pointer for instances of `gadget::Event`, which is in turn a base class for `gadget::KeyEvent` and `gadget::MouseEvent`. Each of these has its own reference-counted smart pointer, namely `gadget::KeyEventPtr` and `gadget::MouseEventPtr`. This seems like a lot of types to understand, but it is simple enough to use by keeping in mind that there are only two types of events: key press events and mouse events. Furthermore, user code should only be interested in `gadget::KeyEventPtr` and `gadget::MouseEventPtr`. The specific event type is determined through the method `gadget::Event::type()`.

At this point, observant readers will be wondering how to downcast instances of `gadget::EventPtr` to either `gadget::KeyEventPtr` or `gadget::MouseEventPtr`. All three of the reference-counted smart pointer types make use of Boost [http://www.boost.org/] shared pointers (instantiations of the type `boost::shared_ptr<T>`), part of the Boost smart pointer library [http://www.boost.org/libs/smart_ptr/smart_ptr.htm]. Boost shared pointers have their own version of the built-in C++ operation `dynamic_cast<T,U>()` called `boost::dynamic_pointer_cast<T,U>()`. It works the same way as `dynamic_cast<T,U>()`, but it is designed specifically for Boost shared pointers. In Example 3.2, "Using `gadget::KeyboardMouseInterface` in an Application Object", we see how to put all of this together in order to handle keyboard and mouse input.

Example 3.2. Using `gadget::KeyboardMouseInterface` in an Application Object

```
 1 #include <boost/shared_ptr.hpp>
   #include <gadget/Type/KeyboardMouseInterface.h>
   #include <gadget/Type/KeyboardMouse/KeyEvent.h>
   #include <gadget/Type/KeyboardMouse/MouseEvent.h>
 5 #include <vrj/Draw/OGL/GlApp.h>


   // This is here to shorten the use of the function in preFrame()
```

[1] Behind the scenes, `gadget::KeyboardMouse::EventQueue` is a typedef for `std::vector<gadget::EventPtr>`.

```
   // boost::dynamic_pointer_cast<T,U>() below.
10 using namespace boost;

   class MyApp : public vrj::GlApp
   {
   public:
15    MyApp() : vrj::GlApp()
      {
         /* Do nothing. */ ;
      }

20    virtual ~MyApp()
      {
         /* Do nothing. */ ;
      }

25    void init()
      {
         mKeyboard.init("VJKeyboard");
      }

30    void preFrame()
      {
         gadget::KeyboardMouse::EventQueue evt_queue =
            mKeyboard->getEventQueue();
         gadget::KeyboardMouse::EventQueue::iterator i;
35
         // Loop over all the keyboard and mouse events that
         // occurred since the last frame.
         for ( i = evt_queue.begin(); i != evt_queue.end(); ++i )
         {
40          const gadget::EventType type = (*i)->type();

            if ( type == gadget::KeyPressEvent ||
                 type == gadget::KeyReleaseEvent )
            {
45             gadget::KeyEventPtr key_evt =
                  dynamic_pointer_cast<gadget::KeyEvent>(*i);
               // Handle the key press event ...
            }
            else if ( type == gadget::MouseButtonPressEvent ||
50                    type == gadget::MouseButtonReleaseEvent ||
                      type == gadget::MouseMoveEvent )
            {
               gadget::MouseEventPtr mouse_evt =
                  dynamic_pointer_cast<gadget::MouseEvent>(*i);
55             // Handle the mouse event ...
            }
         }
      }

60    void draw()
      {
         // Draw something ...
      }

65 private:
      gadget::KeyboardMouseInterface mKeyboard;
   };
```

Using `gadget::PositionInterface`

Position tracking devices return data to application objects as 4×4 transformation matrices. The return type of `gadget::PositionProxy::getData()` is `gmtl::Matrix44f`, which was introduced in the section called "The `gmtl::Matrix44f` Helper Class". All tracking devices return a full transformation matrix even if the physical tracking hardware is only capable of returning translation or orientation data.

Tip

When querying a positional device for its data, it is critical to ask for the data in the units that the application expects. An easy way to do this is to pass the result of `getDrawScaleFactor()` to `gadget::PositionProxy::getData()`. By default, `gadget::PositionProxy::getData()` returns data in feet, and the implementation of `vrj::App::getDrawScaleFactor()` returns `gadget::PositionUnitConversion::ConvertToFeet`. This is for backwards compatibility with VR Juggler 1.0 behavior. See Example 3.3, "Requesting Positional Data in Application-Specific Units" for an example of this. Refer to the section called "`vrj::App::getDrawScaleFactor()`" for more information about `getDrawScaleFactor()`.

Example 3.3. Requesting Positional Data in Application-Specific Units

```
#include <gmtl/Matrix.h>

#include <gadget/Type/Position/PositionUnitConversion.h>
#include <gadget/Type/PositionInterface.h>

#include <vrj/Draw/OGL/GlApp.h>


class MyApp : public vrj::GlApp
{
public:
   MyApp() : vrj::GlApp()
   {
      /* Do nothing. */ ;
   }

   virtual ~MyApp()
   {
      /* Do nothing. */ ;
   }

   void init()
   {
      mWand.init("VJWand");
   }

   // Use meters for the application units.
   float getDrawScaleFactor()
   {
      return gadget::PositionUnitConversion::ConvertToMeters;
   }

   void preFrame()
   {
      // Request the current wand transformation matrix in
      // application units (meters).
      const float units = getDrawScaleFactor();
      gmtl::Matrix44f wand_mat(mWand->getData(units));
```

```
         // Do something with the wand transformation ...
40    }

      void draw()
      {
         // Draw something ...
45    }

   private:
      gadget::PositionInterface mWand;
   };
```

Using `gadget::StringInterface`

String (text- or word-driven) devices were introduced in Gadgeteer 1.0 Beta 1. They are an evolving device type geared towards textual or spoken input. In Gadgeteer 1.0, such input comes in the form of spoken phrases that are returned to the application object as strings matching those in a pre-defined grammar. In future versions of Gadgeteer, this device type may be used for additional forms of high-level input. The return type of `gadget::StringProxy::getData()` is `std::string`, and it is up to the person configuring the string device to set up the recognized grammar.

Stupefied Proxies

The indirection provided by device proxies facilitates run-time reconfiguration of hardware devices. If a hardware device breaks down while an application is running, the device can be replaced without shutting down and restarting the application. To support this capability, proxies can become "stupefied," which means that they are not connected to a device and cannot return new data.

In general, application programmers do not need to worry about stupefied proxies. Data will be returned by the proxy whether it is stupefied or not, but if the proxy is stupefied, it cannot return *new* data. Stupefied proxies can occur as a result of an error in the VR Juggler configuration or because the hardware device pointed at by the proxy failed to start up correctly.

To determine whether a proxy is stupefied, the method `gadget::Proxy::isStupefied()` can be used. The stupefied state cannot be changed programatically by user code, however. Only the Input Manager is capable of reconfiguring a proxy to point at a new valid device. Hence, changing the stupefication state of a proxy from an application object will have no effect and may cause the application to crash.

Warning

In all versions of VR Juggler prior to 2.0 Beta 3, the word "stupefied" was misspelled as "stupified." The method `gadget::Proxy::isStupified()` is retained in Gadgeteer 1.0 for backwards compatibility, but it will be removed in Gadgeteer 1.2 (which will ship as part of VR Juggler 2.2).

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>` provides the method interface for all the type-specific instantiations, including the overloaded dereference operator. The base class of `gad-`

`get::DeviceInterface<T>`, `gadget::BaseDeviceInterface`, maintains the name of the proxy and the proxy reference itself, and it provides the all-important `init()` method.

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.

Part II. Application Programming

In each chapter of this part, we outline specific areas of interest for application developers. We begin with the basics of writing VR Juggler applications in general. Then, we move on to the use of different graphics programming interfaces, such as OpenGL and OpenSG, with VR Juggler. Afterwards, we address some additional, but vital, topics such as how to write cluster-capable applications and how to add sound to VR Juggler applications. This part concludes with chapters providing tips on porting applications written using other toolkits such as GLUT and the CAVElibs™.

Chapter 4. Application Authoring Basics

In this chapter, we build upon the information presented in the previous chapters to explain the basics of writing VR Juggler applications. This information will apply to all applications regardless of the graphics programming interface used to render the immersive space. Understanding this chapter will be critical in being able to write effective, portable VR Juggler applications.

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.

Basic Application Information

As described in previous chapters (see Chapter 1, *Getting Started*, for example), 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:

```
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.

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:

```
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.

Getting Input

There are many types of input devices that VR Juggler application objects can use including positional, digital, and analog. All application objects 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 proxies 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. In this section, we will only demonstrate the use of position and digital device interfaces. Refer to the section called "Device Proxies and Device Interfaces" for more detailed information on the use of all the available device interfaces.

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.

```
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.

```
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.

```
...
const float units = getDrawScaleFactor();
gmtl::Matrix44f wand_pos(mWand->getData(units));
...
```

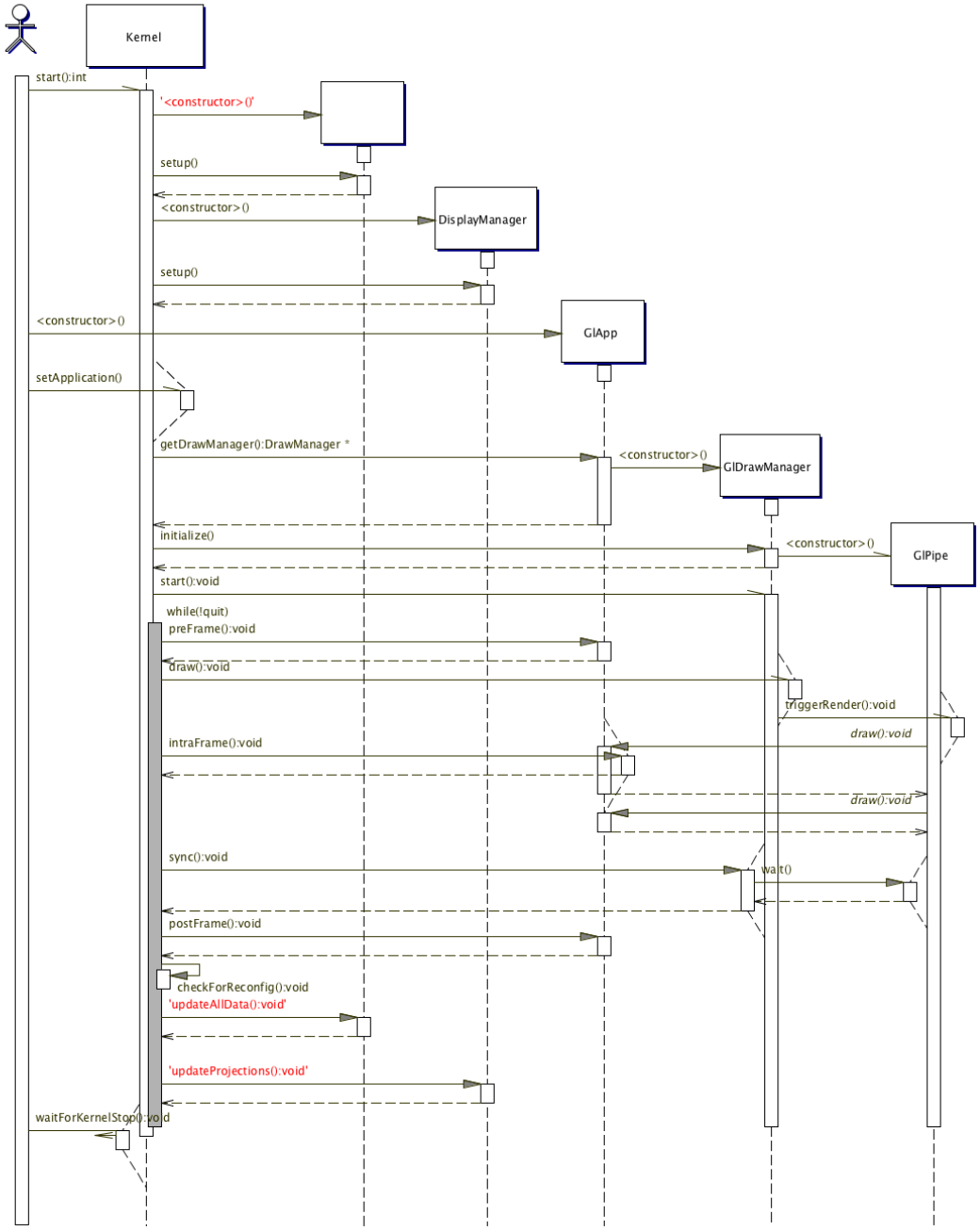
The above code shows an example of using the positional device interface in an application. It shows some sample code where the application copies the positional information from a device interface. When it is dereferenced, the device interface figures out what device it points to and returns the data from that device. Again, refer to the section called "Device Proxies and Device Interfaces" for more information about using device proxies and device interfaces.

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 4.1, “VR Juggler kernel control loop”.

Figure 4.1. 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 `vrj::App::preFrame()`, and
2. Any time spent in `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 `preFrame()` member function. The second point is equally important because it says why user applications should not waste any time in `preFrame()`. Any time spent in `preFrame()` increases system latency and in turn decreases the perceived quality of the environment. Hence, it is crucial to avoid placing computations in `preFrame()`.

Tutorial: Getting Input

In this section, we present a tutorial that demonstrates simple input handling using Gadgeteer device interfaces. The tutorial overview is as follows:

- Description: Simple application that prints the location of the head and the wand.
- Objective: Understand how to get positional and digital input in a VR Juggler application.
- Member functions: `vrj::App::init()`, `vrj::App::preFrame()`
- Directory: `$VJ_BASE_DIR/share/samples/OGL/simple/simpleInput`
- Files: `simpleInput.h`, `simpleInput.cpp`

Class Declaration and Data Members

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

```
1 class simpleInput : public vrj::GlApp
  {
  public:
     virtual void init();
5    virtual void preFrame();

  public:
     gadget::PositionInterface mWand;     // Positional interface for Wand posit
     gadget::PositionInterface mHead;     // Positional interface for Head posit
10   gadget::DigitalInterface  mButton0;  // Digital interface for button 0
     gadget::DigitalInterface  mButton1;  // Digital interface for button 1
  };
```

Initializing the Device Interfaces: `vrj::App::init()`

The devices are initialized in the `init()` member function of the application. For each device interface member variable, the application calls the variable's own `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 *handles* to the named device.

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

Examining the Device Data: `vrj::App::preFrame()`

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

```
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:"
             << " 0:" << mButton0->getData()
             << " 1:" << mButton1->getData()
             << std::endl;

   // -- Get Wand matrix --- //
   const float units = getDrawScaleFactor();
   gmtl::Matrix44f wand_matrix(mWand->getData(units));
   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.

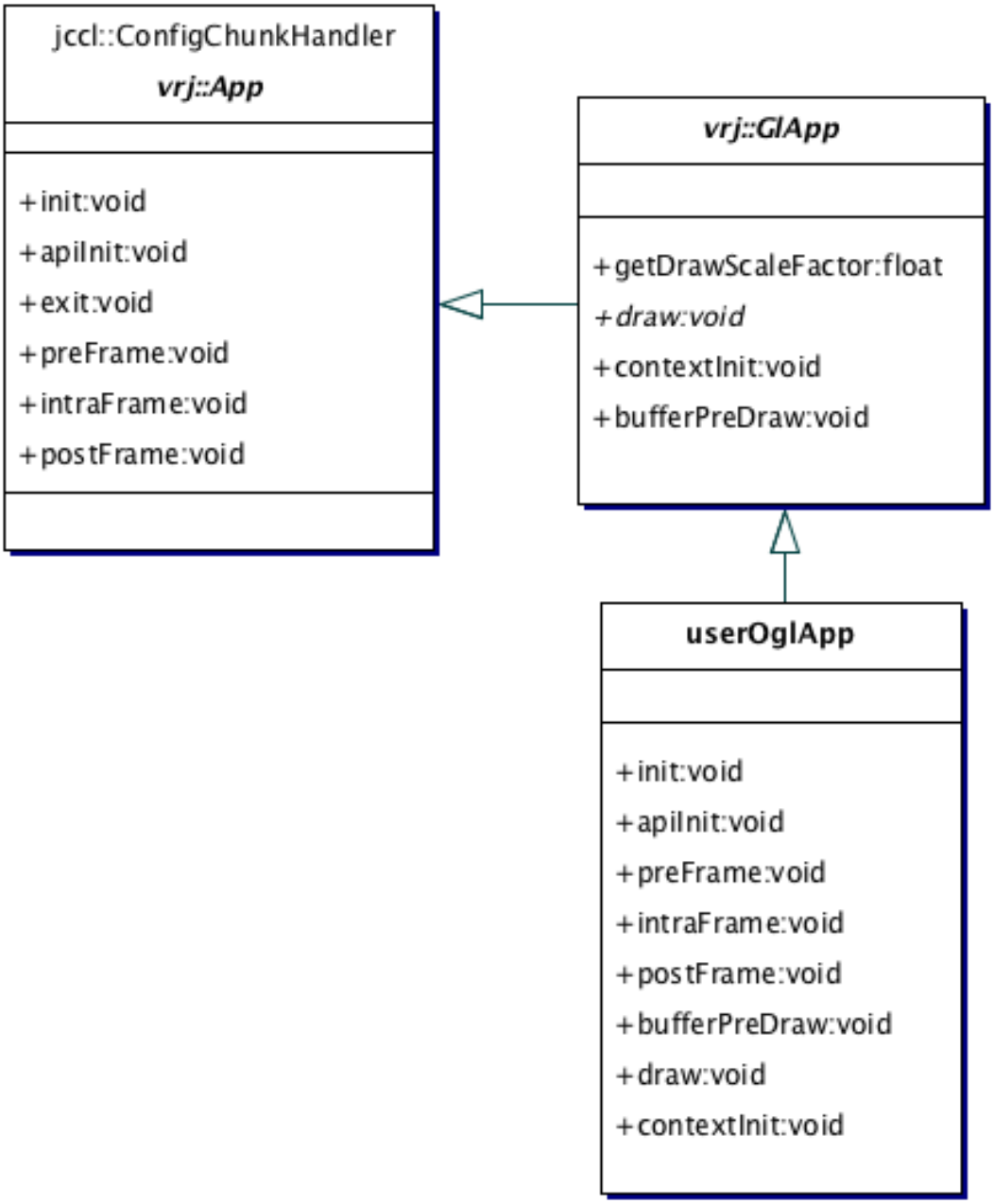
Chapter 5. Using Graphics Programming Interfaces

This chapter comprises the bulk of information about application development. This makes sense when one considers the importance of computer grahpics in the context of immersive applications. In each section of this chapter, we explain the use of different graphics application programming interfaces (APIs) within the scope of VR Juggler. While the sections of this chapter are tied to specific APIs, we highly recommend that all prospective programmers of VR Juggler applications read the first section about OpenGL applications. This section covers core fundamentals of the VR Juggler OpenGL Draw Manager that apply to the use of Open Scene Graph and OpenSG with VR Juggler.

OpenGL Applications

We can now describe how to write OpenGL applications in VR Juggler. An OpenGL-based VR Juggler application must be derived from `vrj::GlApp`. This in turn is derived from `vrj::App`. As was discussed in the application object section, `vrj::App` defines the base interface that VR Juggler expects of all applications. The `vrj::GlApp` class extends this interface by adding members that the VR Juggler OpenGL Draw Manager needs to render an OpenGL application correctly.

Figure 5.1. `vrj::GlApp` application class



In Figure 5.1, "`vrj::GlApp` application class", 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.

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:

```
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:

```
void userApp::draw()
{
   glClear(GL_DEPTH_BUFFER_BIT);

   // Rendering the scene ...
}
```

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 object.

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.

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.

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.

Tutorial: Drawing a Cube with OpenGL

In this section, we present a tutorial that demonstrates simple rendering with OpenGL calls. The tutorial overview is as follows:

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

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.

```
 1 using namespace vrj;
   using namespace gadget;

   class simpleApp : public GlApp
 5 {
   public:
      simpleApp();
      virtual void init();
      virtual void draw();
10
   public:
      PositionInterface mWand;
      PositionInterface mHead;
      DigitalInterface mButton0;
15    DigitalInterface mButton1;
   };
```

The `draw()` Member Function

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

```
 1 using namespace gmtl;

   void simpleApp::draw()
   {
 5    ...
      // Create box offset matrix
      Matrix44f box_offset;
      const EulerAngleXYZf euler_ang(Math::deg2Rad(-90.0f), Math::deg2Rad(0.0f),
                                     Math::deg2Rad(0.0f));
10    box_offset = gmtl::makeRot<Matrix44f>(euler_ang);
      gmtl::setTrans(box_offset, Vec3f(0.0, 1.0f, 0.0f));
      ...
      glPushMatrix();
         // Push on offset
```

```
15          glMultMatrixf(box_offset.getData());
            ...
            drawCube();
         glPopMatrix();
         ...
20 }
```

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.

Exercise

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

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 the section called “Why it is Needed” or to the section called “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
- Texture objects

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.

```
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.

Context-Specific Variables in VR Juggler

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

```
using namespace vrj;

class userApp : public GlApp
{
public:
   draw();
public:
   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.

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.

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.

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.

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.

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 5.1. Initializing context-specific data

```
 1 void userApp::contextInit()
   {
      // Allocate context specific data
      (*mDispListId) = glGenLists(1);
 5
      glNewList((*mDispListId), GL_COMPILE);
         glScalef(0.50f, 0.50f, 0.50f);
         // Call func that draws a cube in OpenGL
         drawCube();
10    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.

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 applications 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.

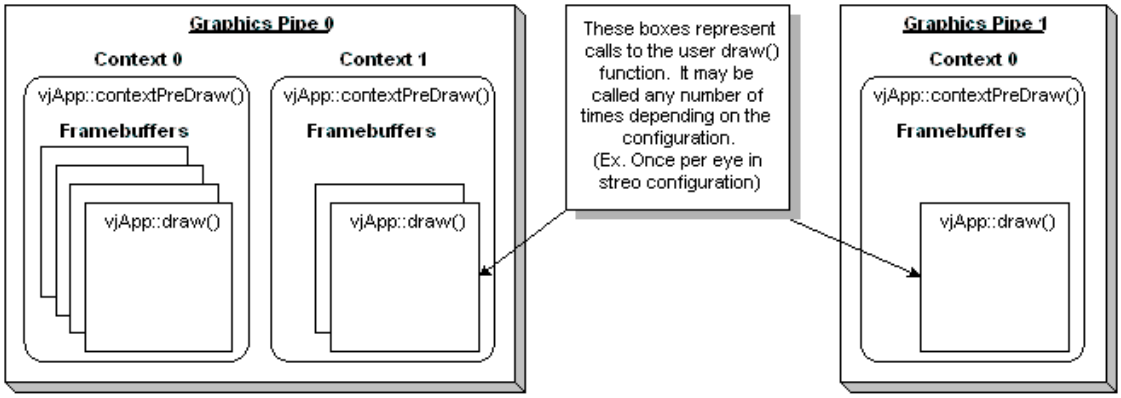
Please notice, however, that since this method is called often and is called in performance-critical areas, you should not do much work in it. Any time taken by this method directly decreases the draw performance of the application. In most cases, we recommend trying to make the function have a very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

```
userApp::contextInit()
{
   if (have work to do)
   {
     // Do it
   }
}
```

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.

Figure 5.2. VR Juggler OpenGL system



Do Not Allocate Context-Specific Data in `draw()`

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.)

Tutorial: Drawing a Cube using OpenGL Display Lists

In this section, we present a tutorial that demonstrates the use of OpenGL display lists with VR Juggler context-specific data. The tutorial overview is as follows:

- Description: Drawing a cube using a display list in the `draw()` member function.
- Objective: Understand how to use context-specific data in an application.
- Member functions: `vrj::App::init()`, `vrj::GlApp::contextInit()`, `vrj::GlApp::draw()`
- Directory: `$VJ_BASE_DIR/share/samples/OGL/simple/contextApp`
- Files: `contextApp.h`, `contextApp.cpp`

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 the section called “Tutorial: Drawing a Cube with OpenGL”. Note the addition of the `contextInit()` declaration and the use of the context-specific data member `mCubeDlId`.

```
1 using namespace vrj;

  class contextApp : public GlApp
  {
5 public:
     contextApp() {;}
```

```
      virtual void init();
      virtual void contextInit();
      virtual void draw();
10     ...
   public:
      // Id of the cube display list
      GlContextData<GLuint> mCubeDlId;
      ...
15 };
```

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 5.1, “Initializing context-specific data”, presented in the section called “Using Context-Specific Data”. That example was based on this tutorial application.

```
 1 void contextApp::contextInit()
   {
      // Allocate context specific data
      (*mCubeDlId) = glGenLists(1);
 5
      glNewList((*mCubeDlId), GL_COMPILE);
         glScalef(0.50f, 0.50f, 0.50f);
         drawCube();
      glEndList();
10    ...
   }
```

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.

```
 1 using namespace gmtl;

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

Exercise

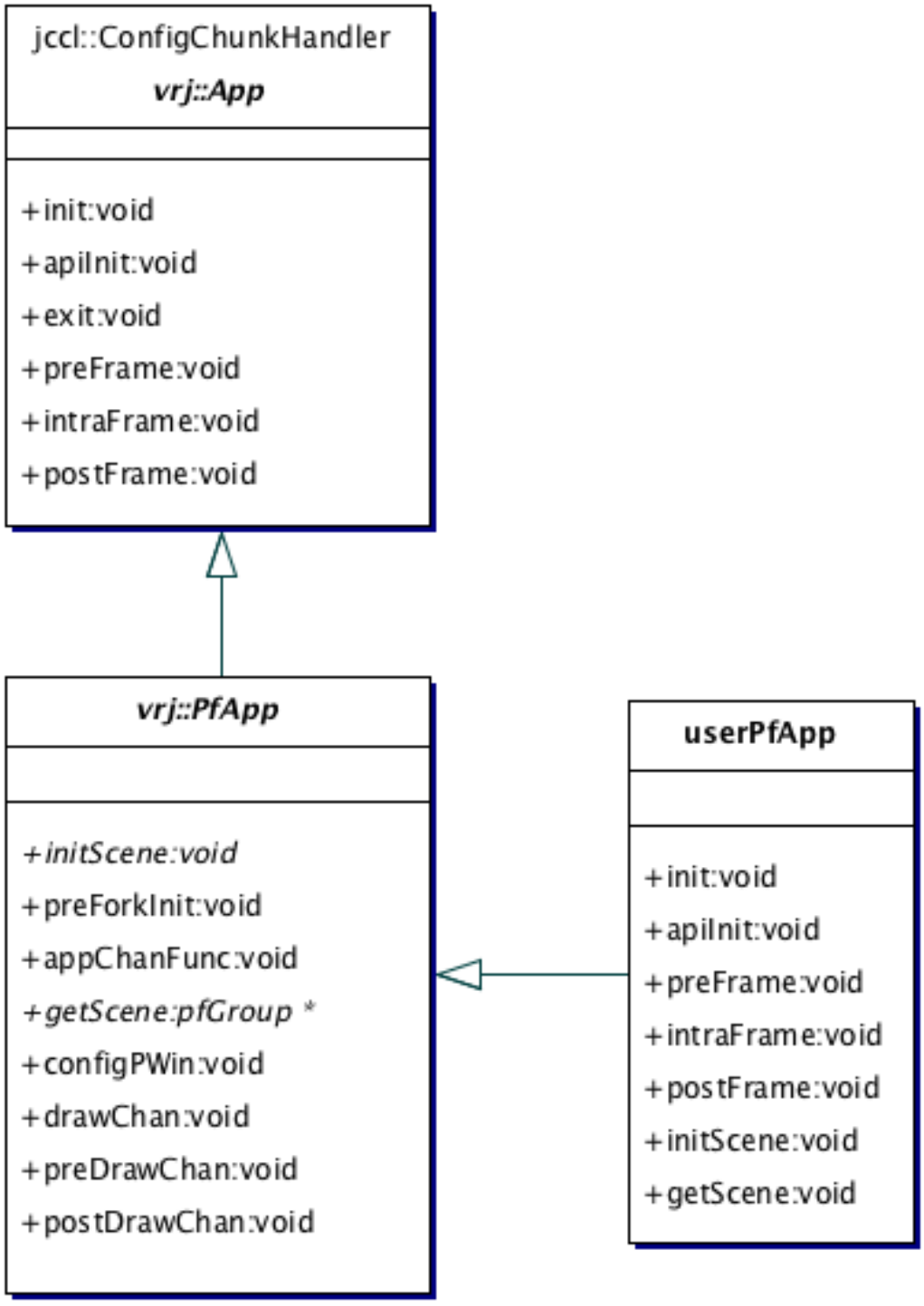
In the tutorial application code, replace the call to `drawAxis()` with a display list call.

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 5.3, "`vrj::PfApp` application class" shows how `vrj::PfApp` fits into the class hierarchy of a Performer-based VR Juggler application.

Figure 5.3. `vrj::PfApp` application class



Two of the methods added to the application interface by `vrj::PfApp` are `initScene()` and `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 `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 tu-

torial application will be provided to illustrate the concepts presented.

Scene Graph Initialization: `vrj::PfApp::initScene()`

In an application using OpenGL Performer, the scene graph must be initialized before it can be used. The method `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, `vrj::PfApp::initScene()` is invoked after the Performer functions `pfInit()` and `pfConfig()` but before `vrj::App::apiInit()`.

Scene Graph Access: `vrj::PfApp::getScene()`

In order for Performer to render the application scene graph, it must get access to the scene graph root. The method `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 `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 `pfScene` object. If it is, then lighting will not work.

Possible Misuses

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

Tutorial: Loading a Model with OpenGL Performer

In this section, we present a tutorial that demonstrates model loading with OpenGL Performer. The tutorial overview is as follows:

- Description: Simple OpenGL Performer application that loads a model.
- Objective: Understand how to load a model, add it to a scene graph, and return the root to VR Juggler.
- Member functions: `vrj::PfApp::initScene()`, `vrj::PfApp::getScene()`
- Directory: `$VJ_BASE_DIR/share/samples/Pf/simple/simplePf`
- Files: `simplePfApp.h`, `simplePfApp.cpp`

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()`.

```
1 class simplePfApp : public vrj::PfApp
  {
```

```
   public:
      simplePfApp();
 5    virtual ~simplePfApp();

      virtual void preForkInit();
      virtual void initScene();
      virtual pfGroup* getScene();
10    void setModel(std::string modelFile);

   public:
      std::string    mModelFileName;

15    pfGroup*       mLightGroup;
      pfLightSource* mSun;
      pfGroup*       mRootNode;
      pfNode*        mModelRoot;
   };
```

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:

```
 1 void simplePfApp::initScene ()
   {
      // Allocate all the nodes needed
      mRootNode = new pfGroup;
 5
      // Create the SUN light source
      mLightGroup = new pfGroup;
      mSun = new pfLightSource;
      mLightGroup->addChild(mSun);
10    mSun->setPos(0.3f, 0.0f, 0.3f, 0.0f);
      mSun->setColor(PFLT_DIFFUSE, 1.0f, 1.0f, 1.0f);
      mSun->setColor(PFLT_AMBIENT, 0.3f, 0.3f, 0.3f);
      mSun->setColor(PFLT_SPECULAR, 1.0f, 1.0f, 1.0f);
      mSun->on();
15
      // --- LOAD THE MODEL -- //
      mModelRoot = pfdLoadFile(mModelFileName.c_str());

      // -- CONSTRUCT STATIC STRUCTURE OF SCENE GRAPH -- //
20    mRootNode->addChild(mModelRoot);
      mRootNode->addChild(mLightGroup);
   }
```


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.
3. 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()`.)
4. Finally, the model and the light source nodes are added as children of the root.

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:

```
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()`.

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.

`preForkInit()`

Prototype:

```
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()`.

`appChanFunc()`

Prototype:

```
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()`).

`configPWin()`

Prototype:

```
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.

`getFrameBufferAttrs()`

Prototype:

```
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.

`drawChan()`

Prototype:

```
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:

```
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.

`preDrawChan()`

Prototype:

```
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()`).

`postDrawChan()`

Prototype:

```
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()`).

`pfExit()`: To Call or Not to Call

The Performer function `pfExit()` poses a problem for VR Juggler applications, and some background information will help ensure that readers understand the consequences of using `pfExit()` (or not). The main issue with `pfExit()` as it relates to VR Juggler is that calling `pfExit()` has the side effect of calling the system function `exit()`, which means that it should be (or has to be) the very last function call of a program. Prior to VR Juggler 2.0.1, the VR Juggler Performer Draw Manager was written to call `pfExit()` from within the method `vrj::PfDrawManager::closeAPI()`. Before VR Juggler 2.0 Beta 3, however, this method of the `vrj::PfDrawManager` interface had never been called—the result of the kernel shutdown process being incomplete. With the more complete kernel shutdown process in VR Juggler 2.0 Beta 3 and 2.0.0, authors of Performer-based VR Juggler applications saw their applications exiting prematurely after invoking the kernel shutdown. More specifically, any code that was intended to be executed after `vrj::PfDrawManager::closeAPI()` would not be executed. Such code includes `vrj::App::exit()` or an override thereof; an application object destructor; and anything else to be done after `vrj::Kernel::waitForKernelStop()` returned.

To remedy this problem, `vrj::PfDrawManager::closeAPI()` in VR Juggler 2.0.1 and newer *does not* call `pfExit()`. Rather, it is the responsibility of the application programmer to call

`pfExit()` if s/he so desires. Failing to call `pfExit()` could result in resource leaks from Performer, but calling `pfExit()` has been known to cause application crashes (irrespective of whether VR Juggler is used). In general, users should call `pfExit()` at the end of their `main()` function, but if doing so causes the application to crash on exit, then not calling `pfExit()` is probably the better option.

> Resources that could be leaked by Performer if `pfExit()` is not called would generally fall under the heading of "temporary files." Resource leaks are not memory leaks since the operating system will automatically deallocate any memory held by Performer at the time of application exit. Since Performer may allocate temporary files for its process locking and shared memory arena, it would usually be best to ensure that Performer cleans up after itself.

The important thing to remember is that the application object destructor needs to be called *before* `pfExit()` is called. Refer to Example 5.2, "Using `pfExit()` with a Heap-Allocated Application Object" for an example of how `pfExit()` would be used with a VR Juggler application object allocated on the heap. For a stack-allocated application object, see Example 5.3, "Using `pfExit()` with a Stack-Allocated Application Object".

Example 5.2. Using `pfExit()` with a Heap-Allocated Application Object

```
 1 int main(int argc, char* argv[])
   {
      vrj::Kernel* kernel = vrj::Kernel::instance();

 5    // Allocate the application object on the heap.  Its
      // destructor will be called manually before calling
      // pfExit().
      simplePfApp* application = new simplePfApp();

10    // Load config files.
      for ( int i = 2; i < argc; ++i )
      {
         kernel->loadConfigFile(argv[i]);
      }
15
      kernel->start();

      // Configure the application.
      application->setModel(argv[1]);
20    kernel->setApplication(application);

      // Wait for the kernel to shut down.
      kernel->waitForKernelStop();

25    // Final application clean-up.
      delete application;

      // Clean up Performer.
      // Calls system exit() function and therefore never returns.
30    pfExit();

      return 0;
   }
```

Example 5.3. Using `pfExit()` with a Stack-Allocated Application Object

```
 1 int main(int argc, char* argv[])
   {
      // Nested scope for stack-allocated data.
      {
 5       vrj::Kernel* kernel = vrj::Kernel::instance();

         // Load config files.
         for ( int i = 2; i < argc; ++i )
         {
10          kernel->loadConfigFile(argv[i]);
         }

         kernel->start();

15       // Allocate the application object on the stack.  Its
         // destructor will be called automatically at the end
         // of this nested scope.
         simplePfApp application;

20       // Configure the application and give it to the kernel.
         application.setModel(argv[1]);
         kernel->setApplication(&application);

         // Wait for the kernel to shut down.
25       kernel->waitForKernelStop();
      }

      // Clean up Performer.
      // Calls system exit() function and therefore never returns.
30    pfExit();

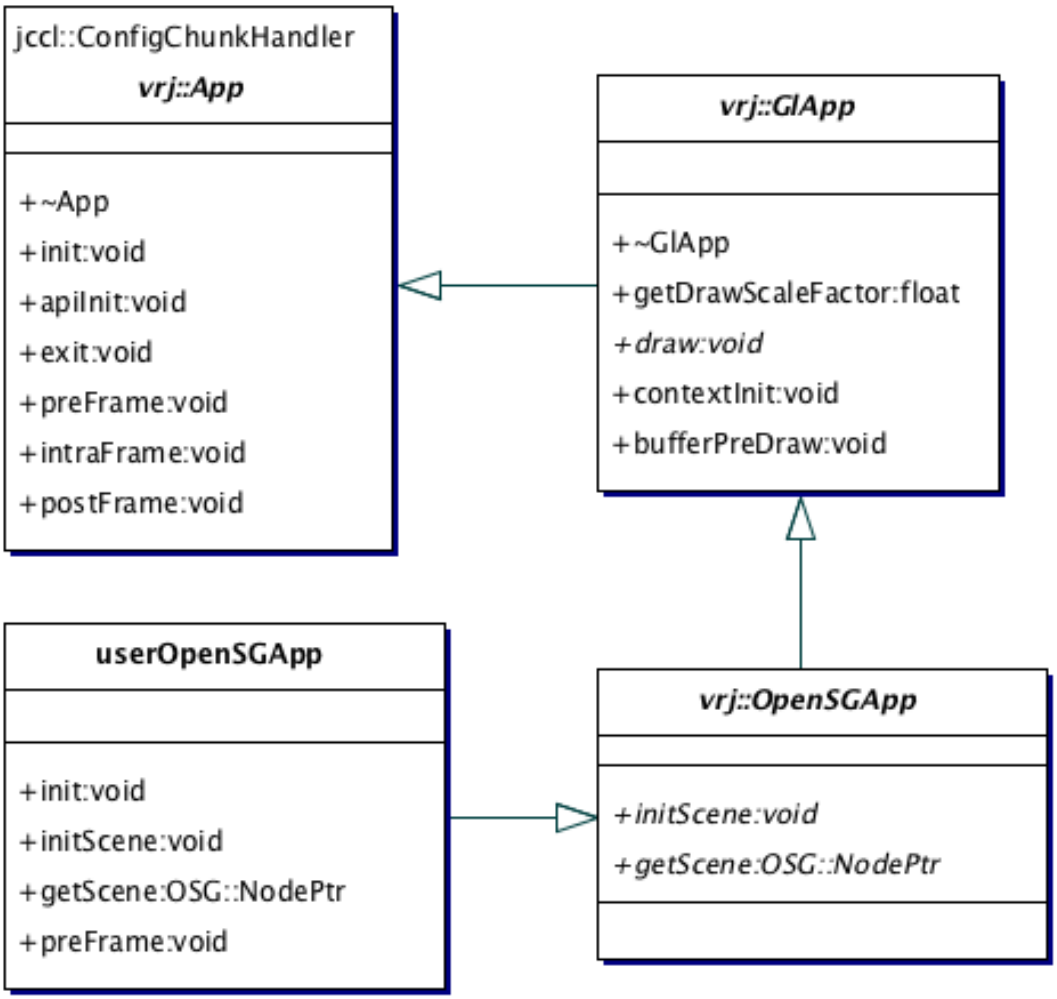
      return 0;
   }
```

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`. `vrj::OpenSGApp` extends `vrj::GlApp` by adding methods that deal with scene graph initialization and access. Figure 5.4, "`vrj::OpenSGApp` application class" shows how `vrj::OpenSGApp` fits into the class hierarchy of an OpenSG-based VR Juggler application.

Figure 5.4. `vrj::OpenSGApp` application class



The two main application methods for `vrj::OpenSGApp` VR Juggler applications are `initScene()` and `getScene()`. 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.

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.

Scene Graph Access: `vrj::OpenSGApp::getScene()`

In order for OpenSG to render the application scene graph, it must get access to the scene graph root. The method `vrj::OpenSGApp::getScene()` 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 (for example when rendering or updating). Since the job of `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 current scene graph root node.

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.

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

To update the scene graph, use either `preFrame()`, `intraFrame()`, or `postFrame()`.

Tutorial: Loading a Model with OpenSG

In this section, we present a tutorial that demonstrates model loading with OpenSG. The tutorial overview is as follows:

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

Class Declaration

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

```
 1 class OpenSGNav : public vrj::OpenSGApp
   {
   public:
      OpenSGNav(vrj::Kernel* kern);
 5    virtual ~OpenSGNav();

      virtual void init();
      virtual void contextInit();
      virtual void preFrame();
10
      virtual void initScene();
      virtual OSG::NodePtr getScene();
```

```
      void setModelFileName(std::string filename);
15
   private:
      void initGLState();

   private:
20    std::string mFileToLoad;

      OSG::NodePtr        mSceneRoot;
      OSG::TransformPtr   mSceneTransform;
      OSG::NodePtr        mModelRoot;
25
      OSG::NodePtr  mLightNode;
      OSG::NodePtr  mLightBeacon;

   public:
30    gadget::PositionInterface  mWandPos;
      gadget::DigitalInterface   mButton0;
      gadget::DigitalInterface   mButton1;
      gadget::DigitalInterface   mButton2;
      float  velocity;
35 };
```

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:

```
 1 void OpenSGNav::initScene()
   {
      // Load the model to use
      if (mFileToLoad == std::string("none"))
 5    {
         mModelRoot = OSG::makeTorus(.5, 2, 16, 16);
      }
      else
      {
10       mModelRoot =
            OSG::SceneFileHandler::the().read(mFileToLoad.c_str());
      }

      // --- Light setup --- //
15    // - 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 =
20       OSG::DirectionalLight::create();
      OSG::TransformPtr light_beacon_core =
         OSG::Transform::create();

      // Setup light beacon
25    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);
30    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   (0.9, 0.8, 0.8, 1);
      light_core->setDiffuse   (0.6, 0.6, 0.6, 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
   mSceneRoot = OSG::Node::create();
   mSceneTransform = OSG::Transform::create();

   // Set the root node
   OSG::beginEditCP(mSceneRoot);
      mSceneRoot->setCore(mSceneTransform);
      mSceneRoot->addChild(mLightNode);
   OSG::endEditCP(mSceneRoot);
}
```






We begin by loading the file set in `OpenSGNav::setModelFileName()`. If no file name was , we default to using a simple torus model. The model object is `mModelRoot`.
ext, we create a node for the light, which we define as a beacon light.
e model is added under the light in the scene graph so that it gets lit.
en, the root note for the scene graph is created. This is what will be returned to OpenSG for ren-ring by `OpenSGNav::getScene()`.
nally, we add the light node as a child of the scene root. Remember that the light node already s the loaded model as a child.

The `getScene()` Member Function

The method `vrj::OpenSGApp::draw()` will call the application's `getScene()` 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:

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

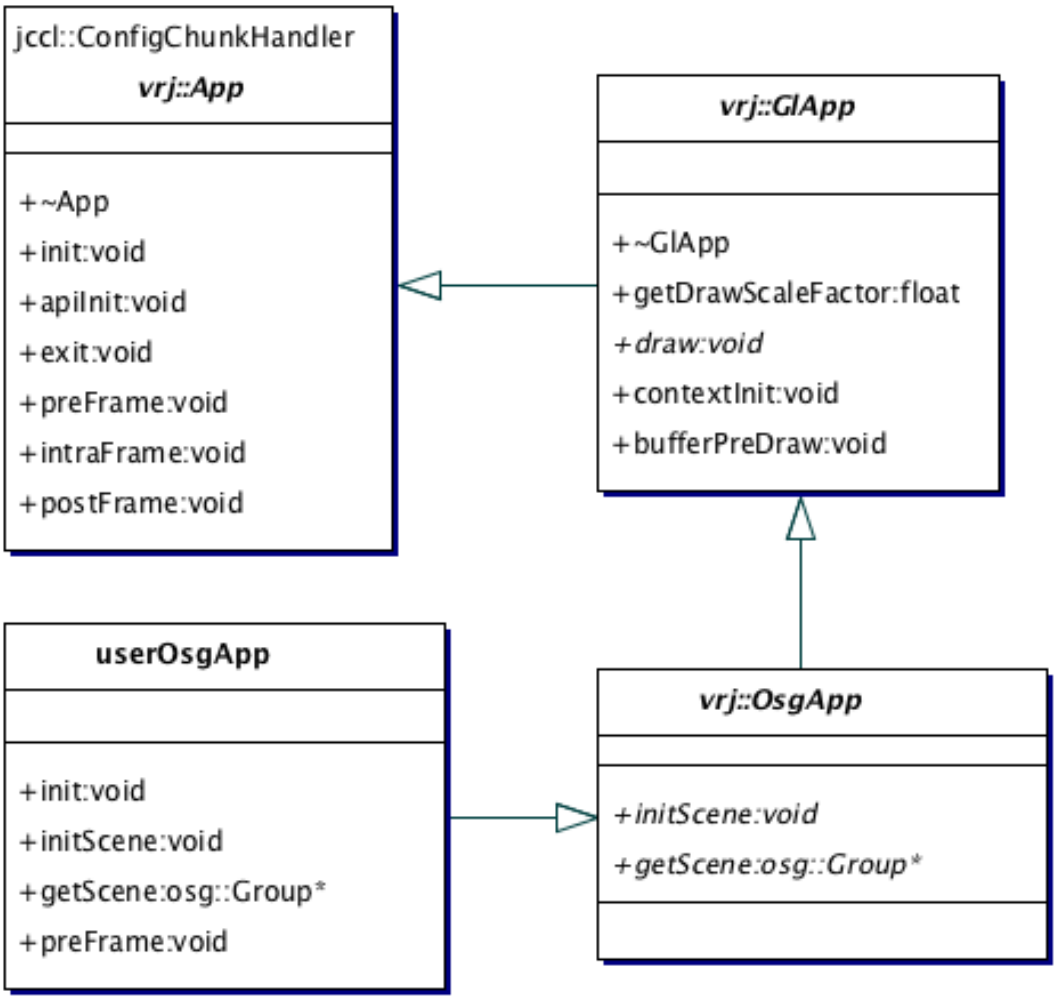
The simplicity of this method implementation is not limited to the simple tutorial from which it is taken. All OpenSG-based VR Juggler applications can take advantage of this idiom where the root node is a member variable returned in `getScene()`.

Open Scene Graph Applications

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

An OSG-based VR Juggler application must derive from `vrj::OsgApp`. The `vrj::OsgApp` class is derived from the `vrj::GlApp` presented previously, which in turn derives from `vrj::App`. `vrj::OsgApp` extends `vrj::GlApp` by adding methods that deal with scene graph initialization and access. Figure 5.5, "`vrj::OsgApp` application class" shows how `vrj::OsgApp` fits into the class hierarchy of an OSG-based VR Juggler application.

Figure 5.5. `vrj::OsgApp` application class



The two main application methods for `vrj::OsgApp` VR Juggler applications are `initScene()` and `getScene()`. These are called by the OSG 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::OsgApp`). The rest of this section gives a more detailed description of these methods and some sample code to illustrate the concepts presented.

Scene Graph Initialization: `vrj::OsgApp::initScene()`

In an application using OSG, the scene graph must be initialized before it can be used. The method `vrj::OsgApp::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.

Important

During the application initialization, `vrj::OsgApp::initScene()` is invoked. This method is invoked in `vrj::OsgApp::init()`. Therefore, user application objects that derive from `vrj::OsgApp` should be sure to invoke `vrj::OsgApp::init()` in their overriding `init()` method, or the contents of the overriding `init()` method should be moved into the implementation of `initScene()`.

Scene Graph Access: `vrj::OsgApp::getScene()`

In order for OSG to render the application scene graph, it must get access to the scene graph root. The method `vrj::OsgApp::getScene()` will be called by the OSG application class wrapper so that it can get access to the currently active scene graph whenever the wrapper needs to use it (for example when rendering or updating). Since the job of `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 current scene graph root node.

Possible Misuses

Do not do any CPU-heavy processing in this method. Because it 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.

```
virtual osg::Group* getScene()
{
   return mSceneRoot;  // Return the root of the graph
}
```

To update the scene graph, use either `preFrame()`, `intraFrame()`, or `postFrame()`.

Tutorial: Loading a Model with Open Scene Graph

In this section, we present a tutorial that demonstrates model loading and scene navigation using Open Scene Graph. The tutorial overview is as follows:

- Description: Simple OSG application that loads a model and allows navigation.
- Objectives: Understand how to load a model, add it to a scene graph, and return the root to VR Juggler.
- Member functions: `vrj::OsgApp::initScene()`, `vrj::OsgApp::getScene()`
- Directory: `$VJ_BASE_DIR/share/vrjuggler/samples/OSG/simple/osgNav`
- Files: `OsgNav.h`, `OsgNav.cpp`

Class Declaration

The following application class is called `osgNav`. It is derived from `vrj::OsgApp` and has custom `initScene()`, `getScene()`, `configSceneView()`, `preFrame()`, and `latePreFrame()` methods declared. Refer to `OsgNav.h` for the implementation of `setModelFileName()`. Note that we will ignore the code for remote navigation via a Tweek-based GUI in this description.

```
 1 class OsgNav : public vrj::OpenSGApp
   {
   public:
      OsgNav(vrj::Kernel* kern, int& argc, char** argv);
 5    virtual ~OsgNav();

      virtual void configSceneView();

      virtual void initScene();
10    void myInit();
      virtual osg::Group* getScene();

      virtual void preFrame();
      virtual void latePreFrame();
15
      void setModelFileName(std::string filename);

   private:
      osg::Group*             mRootNode;
20    osg::Group*             mNoNav;
      osg::MatrixTransform*   mNavTrans;
      osg::MatrixTransform*   mModelTrans;
      osg::Node*              mModel;

25    OsgNavigator  mNavigator;

      std::string   mFileToLoad;

      vpr::Interval mLastPreFrameTime;
30
   public:
      gadget::PositionInterface  mWand;
      gadget::PositionInterface  mHead;
      gadget::DigitalInterface   mButton0;
35    gadget::DigitalInterface   mButton1;
      gadget::DigitalInterface   mButton2;
      gadget::DigitalInterface   mButton3;
      gadget::DigitalInterface   mButton4;
      gadget::DigitalInterface   mButton5;
40 };
```

The `initScene()` Member Function

The implementation of `initScene()` is in `OsgNav.cpp`. This method looks very similar to the usual implementation of init() in other application object examples. The important thing to note is the last line of the method body where myInit() is invoked.

```
 1 void OsgNav::initScene()
   {
      mWand.init("VJWand");
      mHead.init("VJHead");
 5    mButton0.init("VJButton0");
      mButton1.init("VJButton1");
```

```
      mButton2.init("VJButton2");
      mButton3.init("VJButton3");
      mButton4.init("VJButton4");
10    mButton5.init("VJButton5");

      myInit();
   }
```

Within the `myInit()` method, we see the real work for initializing the scene. In this method, we create the scene graph root node, the lighting node, and load a user-specified model. The implementation, found in `OsgNav.cpp`, follows:

```
void OsgNav::myInit()
{
   //
   //          /-- mNoNav
   // mRootNode
   //         \-- mNavTrans -- mModelTrans -- mModel

   //The top level nodes of the tree
   mRootNode = new osg::Group();
   mNoNav    = new osg::Group();
   mNavTrans = new osg::MatrixTransform();

   mNavigator.init();

   mRootNode->addChild(mNoNav);
   mRootNode->addChild(mNavTrans);

   //Load the model
   std::cout << "Attempting to load file: "
             << mFileToLoad << "... " << std::flush;
   mModel = osgDB::readNodeFile(mFileToLoad);
   std::cout << "done." << std::endl;

   // Transform node for the model
   mModelTrans  = new osg::MatrixTransform();
   //This can be used if the model orientation needs to change
   mModelTrans->preMult(
      osg::Matrix::rotate(gmtl::Math::deg2Rad(-90.0f),
                          1.0f, 0.0f, 0.0f)
   );

   if(NULL == mModel)
   {
      std::cout << "ERROR: Could not load file: "
                << mFileToLoad << std::endl;
   }
   else
   {
      // Add model to the transform
      mModelTrans->addChild(mModel);
   }

   // Add the transform to the tree
   mNavTrans->addChild(mModelTrans);

   // run optimization over the scene graph
   osgUtil::Optimizer optimizer;
   optimizer.optimize(mRootNode);
}
```







1. We begin by creating the nodes that will make up the application scene graph. Note that this scene graph will contain two branches: one for nodes that will be affected by user navigation and one for nodes that will not.
2. Next, we attempt to load the model provided through an earlier call to `setModelFileName()`. In this application, the user must identify the model to load on the command line when running the application. The code for handling this can be found in the `main()` function.
3. The model will be attached to the scene graph under a transform node, so we must create that node next.
4. If the model was loaded successfully (`mModel` is not `NULL`), then we attach it to the scene graph under the freshly created model transform node.
5. Whether the named model was loaded successfully or not, we attach the model transform to the scene graph under the navigation-enabled branch. When the model is loaded successfully, this allows the user to fly around the model.
6. Finally, we use `osgUtil::Optimizer` to optimize the scene graph that we have built up.

The getScene() Member Function

The method `vrj::OsgApp::draw()` will call the application's `getScene()` method to get the root of the scene graph. The implementation of this method can be found in `OsgNav.h`. The code is as follows:

```
osg::Group* OsgNav::getScene()
{
   return mRootNode;
}
```

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

VTK Applications

Chapter 6. Additional Application Programming Topics

We now present topics that will be of interest to VR Juggler programmers in general but are not as low-level as those topics described in Chapter 4, *Application Authoring Basics*. Furthermore, understanding the use of graphics APIs within a VR Juggler application will help with understanding how these additional features can be used effectively. As such, it is expected that readers of this chapter will have already read and understood the topics presented in the previous chapters of this part of the book.

Cluster Application Programming

Traditionally, multi-screen immersive systems have relied upon dedicated high-end shared memory graphics workstations or supercomputers to generate interactive virtual environments. These multi-screen immersive systems typically require one or two video outputs for each screen and simultaneously utilize several interaction devices. In recent years this trend of almost exclusively using high-end systems has started to change as commodity hardware has become a viable alternative to high-end systems.

Current technologies have empowered PC-based systems with high-quality graphics hardware, significant amount of memory and computing power, as well as support for many external devices. Their application to virtual reality applications is motivated by the dramatic cost decrease they represent and by the wide range of options and availability. To drive a multi-screen immersive environment we need multiple commodity systems working as a single unit, that is, a tightly synchronized cluster. The challenge is that, although the base technology is standard off-the-shelf technology, there is a lack of software for weaving together the cluster into a platform that supports the creation of virtual environments. Furthermore, there is an even greater lack of software that can allow existing virtual environment designed for high-end system to transparently migrate to a cluster.

In this section, we review the clustering capabilities of VR Juggler 2.0. The current implementation of clustering in VR Juggler is the result of the hard work of many people and of several design and implementation iterations. It is the most important new feature in VR Juggler 2.0, and it is also the most complex new feature internally. At the level of the application object, the clustering infrastructure is largely hidden. Those pieces that are exposed have been designed to be easy to use and to work in non-cluster configurations. This aids in application portability between VR system configurations.

Shared Input Data

One approach to implementing clustering for interactive graphics applications is to share all data received from input devices. This is based on an assumption that the interaction with the computer graphics will occur through input devices such as 6DOF trackers, pointing devices, etc. Using this approach, a distinct copy of the VR application is run on each cluster node, but they all see the same input data. Since changes to the scene are based on information from the input devices, all the nodes will make the same state changes each frame and therefore remain synchronized.

This capability is implemented through the Gadgeteer Remote Input Manager and the `RIMPlugin` used by the Cluster Manager. The basic goal of these components is to provide a distributed shared memory system for VR input device data. Through shared input data, applications can migrate transparently between shared memory VR systems and PC cluster VR systems.

Shared input data is the easiest VR Juggler clustering feature that can be utilized by application object programmers. In simple cases, nothing about an application will have to change to take advantage of shared input data because the details are hidden within the VR Juggler configuration. The Cluster Manager simply needs to be configured to load the `RIMPlugin` and the Start Barrier Plug-in (`StartBarrierPlugin`) to enable applications to take advantage of shared input data.

Application-Specific Shared Data

When input data sharing is not sufficient to enable a VR Juggler application to run on a cluster, the next option is to use application-specific shared data. Using this option, VR application developers can easily exchange any type of data across a cluster of machines. For example, we might have a GUI running on a hand-held device that interacts with the VR application to control it. We cannot expect this GUI to connect to all nodes in the cluster. Instead, the GUI connects to a single node that accepts the commands from the GUI and then relays them to the rest of the cluster nodes using application-specific shared data.

Application-specific shared data is implemented through the generic (templated) container type `cluster::UserData<T>` and the Application Data Manager plug-in (`ApplicationDataManager` in the Cluster Manager config element plug-in list). As is typically the case with generic containers, any sort of data can be stored and therefore shared. The only caveat is that the contained type must have the following two methods:

1.
```
void writeObject(vpr::ObjectWriter* writer);
```

2.
```
void readObject(vpr::ObjectReader* reader);
```

Respectively, these two methods are used for serializing and de-serializing shared data types. The simplest way to achieve this is to create a data structure that is a subclass of the abstract type `vpr::SerializableObject` and overriding its pure virtual `writeObject()` and `readObject()` methods.

Deriving from `vpr::SerializableObject` is not viable in all cases, however. If the data that must be shared is of a type defined in a third-party library, it cannot be modified to derive from `vpr::SerializableObject`. In that case, the type `vpr::SerializableObjectMixin<T>` can be used. The methods `vpr::SerializableObjectMixin<T>::writeObject()` and `vpr::SerializableObjectMixin<T>::readObject()` must be specialized for the desired type T.

In either case, the end result is a means to serialize the data to be shared across the cluster, and application object programmers will implement methods named `writeObject()` and `readObject()`. When `writeObject()` is invoked, it is passed a pointer to a `vpr::ObjectWriter` object. An object writer is a simple wrapper around an expandable block of memory. Each write operation appends some number of bytes to the memory block based on the size of the data written. The type `vpr::ObjectWriter` provides methods for writing all the basic C++ data types (int, float, bool, char, etc.), though they are named based on the cross-platform type identifiers provided by the VR Juggler Portable Runtime (vpr::Int32, vpr::Uint8, etc.). Byte ordering (endian) issues are handled internally by the object writer. The implementation of `writeObject()` for any shared data type simply copies the data members of the shared data structure into the object writer.

Inversely, the implementation of `readObject()` reads data from an object reader (an instance of `vpr::ObjectReader`) into the local copy of the data structure. The object reader contains a fixed-size block of memory and a pointer to the current location in that memory block. Each read operation moves the pointer some number of bytes in the memory block based on the size of the data read.

Important

Due to the symmetric nature of `writeObject()` and `readObject()`, the reading and writing of data must occur in the same order. That is, the implementation of `writeObject()` will write the shared data in some order, and `readObject()` must read the shared data back out in the same order.

We now present two examples of using the serializable object concept. The first demonstrates the case when a new data structure can be created; the second is the case when a third-party type must be made serializable. When we make a new data structure, it is quite easy to enable serialization. Consider the basic type shown in Example 6.1, “Declaration of a Serializable Type”. It derives from `vpr::SerializableObject` and overrides `writeObject()` and `readObject()` just as it must. It has three data members of different types that define the state of an instance of our type. The serialization and de-serialization implementation, which is quite straightforward, is shown in Example 6.2, “Serializing an Application-Specific Type”.

Example 6.1. Declaration of a Serializable Type

```
#include <vpr/IO/SerializableObject.h>

class MyType : public vpr::SerializableObject
{
public:
   void writeObject(vpr::ObjectWriter* writer);
   void readObject(vpr::ObjectReader* reader);
   // Other public methods ...

private:
   unsigned int mIntData;
   char         mByteData;
   float        mFloatData;
};
```

Example 6.2. Serializing an Application-Specific Type

```
void MyType::writeObject(vpr::ObjectWriter* writer)
{
   writer->writeUint32(mIntData);
   writer->writeInt8(mByteData);
   writer->writeFloat(mFloatData);
}

void MyType::readObject(vpr::ObjectReader* reader)
{
   mIntData   = reader->readUint32();
   mByteData  = reader->readInt8();
   mFloatData = reader->readFloat();
}
```

Now we consider the case when we need to serialize a third-party type. First, let us assume that we have a type, called `SomeType`, defined in a header file from a third-party C++ library. This is shown in Example 6.3, “Sample Third-Party Type”. The type has three accessor methods for reading its data and three for writing. We can then specialize the methods of `vpr::SerializableObjectMixin<T>` as shown in Example 6.4, “Serializing a Third-Party Type Using `vpr::SerializableObjectMixin<T>`”.

Example 6.3. Sample Third-Party Type

```
class SomeType
{
public:
   unsigned int getIntData();
   void setIntData(unsigned int v);

   char getByteData();
   void setByteData(char v);

   float getFloatDat();
   void setFloatData(float v);

private:
   // Private data ...
};
```

Example 6.4. Serializing a Third-Party Type Using `vpr::SerializableObjectMixin<T>`

```
#include <vpr/IO/SerializableObject.h>

template<>
void vpr::SerializableObjectMixin<SomeType>::
   writeObject(vpr::ObjectWriter* writer)
{
   writer->writeUint32(getIntData());
   writer->writeInt8(getByteData());
   write->writeFloat(getFloatData());
}

template<>
void vpr::SerializableObjectMixin<SomeType>::
   readObject(vpr::ObjectReader* reader)
{
   setIntData(reader->readUint32());
   setByteData(reader->readInt8());
   setFloatData(reader->readFloat());
}
```

The magic of `vpr::SerializableObjectMixin<T>` allows the specialized methods to behave as member functions in `SomeType`. This means that the specialized members have easy access to all public and protected members of `SomeType`.

Now that we have data serialization out of the way, we can turn our attention to the use of `cluster::UserData<T>`, the special type that automates application-specific data sharing. For each type of shared data, the application object will have at least one instance of `cluster::UserData<T>`. Example instantiations of `cluster::UserData<T>` are shown in Example 6.5, “Declaring Instances of `cluster::UserData<T>`”.

Example 6.5. Declaring Instances of `cluster::UserData<T>`

```
#include <vrj/Draw/OGL/GlApp.h>
#include <plugins/ApplicationDataManager/UserData.h>
#include <SomeType.h>
```

```
#include "MyType.h"

class AppObject : public vrj::GlApp
{
public:
   void init();
   void preFrame();
   void latePreFrame();
   void draw();

   // Other public member functions ...

private:
   cluster::UserData<MyType> mMyTypeObj;
   cluster::UserData< vpr::SerializableObjectMixin<SomeType> > mSomeTypeObj;
};
```

Next, we must initialize the `cluster::UserData<T>` instances so that the Application Data Manager plug-in can identify the shared data types and so that the application can determine which cluster node will be allowed to write to the shared data. While there are two ways to do this, we will show only the recommended approach here. First, a globally unique identifier (GUID) must be defined for each and every shared data type instance. The command-line utility **uuidgen** is available on most operating systems for generating new GUIDs (also known as universally unique identifiers or UUIDs). These will be used in the application object `init()` method, as shown in Example 6.6, “Initializing Application-Specific Shared Data”.

Example 6.6. Initializing Application-Specific Shared Data

```
void AppObject::init()
{
   vpr::GUID mytype_guid("99CFD306-32AB-11D9-A963-000D933B5E6A");
   mMyTypeObj.init(mytype_guid);

   vpr::GUID sometype_guid("A154B8E8-32AB-11D9-B4C9-000D933B5E6A");
   mSomeTypeObj.init(sometype_guid);
}
```

Important

Do not use the member function `vpr::GUID::generate()` to initialize the type-specific GUID objects. This will result in every cluster node always having a different GUID value every time the application is run (because GUIDs are unique by definition). If this happens, the Application Data Manager plug-in will never be able to complete its initialization, and the application frame loop will not be able to start on all the cluster nodes.

In conjunction with this, two config elements need to be created. These will be used by the Application Data Manager plug-in to identify which cluster node will be the writer node. The “guid” properties must match the string values used to initialize the `vpr::GUID` objects in Example 6.6, “Initializing Application-Specific Shared Data”. The “hostname” properties set the name of the cluster node that will be the shared data writer. An example of this is shown in Example 6.7, “Application-Specific Shared Data Configuration”.

Example 6.7. Application-Specific Shared Data Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings configuration.version="3.0"?>
<configuration
   xmlns="http://www.vrjuggler.org/jccl/xsd/3.0/configuration"
   name="Example Shared Application Data Configuration"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.0/configuration http://

   <elements>
      <application_data name="MyType Shared Data" version="1">
         <guid>99CFD306-32AB-11D9-A963-000D933B5E6A</guid>
         <hostname>machine1</hostname>
      </application_data>
      <application_data name="SomeType Shared Data" version="1">
         <guid>A154B8E8-32AB-11D9-B4C9-000D933B5E6A</guid>
         <hostname>machine1</hostname>
      </application_data>
   </elements>
</configuration>
```

Caution

Be very careful to ensure that the GUID strings match correctly. This means matching the strings in the `application_data` config element "guid" property with the use in the application code. If the GUID strings are not matched correctly, the Application Data Manager will not be able to match the objects initialized in the application object `init()` method.

Now that the shared data is initialized and ready to use, we can write to and read from it—the Application Data Manager plug-in will take care of the rest. Only one node can be allowed to write to the data. This is determined through the use of the method `cluster::UserData<T>::isLocal()`. This method returns a Boolean value that indicates whether the data is "local." The local node is the one named in the configuration element, as shown earlier. Writes to shared data should only occur in `preFrame()` or `postFrame()` after testing the result of `cluster::UserData<T>::isLocal()`. This is shown in Example 6.8, "Writing to Application-Specific Shared Data".

Important

The `cluster::UserData<T>` instances introduce a level of indirection (using the Smart Pointer design pattern) for accessing the shared data that works in both the cluster and the non-cluster case. No direct access to shared data is allowed when using `cluster::UserData<T>`. This is true both for reading and for writing, demonstrated in Example 6.8, "Writing to Application-Specific Shared Data", in Example 6.9, "Reading from Application-Specific Shared Data in `latePreFrame()`", and in Example 6.10, "Reading from Application-Specific Shared Data in `draw()`".

Example 6.8. Writing to Application-Specific Shared Data

```
void AppObject::preFrame()
{
   if ( mMyTypeObj.isLocal() )
   {
      // Computations ...
      mMyTypeObj->setIntData(...);
      mMyTypeObj->setByteData(...);
      mMyTypeObj->setFloatData(...);
   }
```

```
   if ( mSomeTypeObj.isLocal() )
   {
      // Computations ...
      mSomeTypeObj->setIntData(...);
      mSomeTypeObj->setByteData(...);
      mSomeTypeObj->setFloatData(...);
   }
}
```

After the snapshot of the application-specific shared data for the current frame has been distributed to the cluster nodes, it is time to read the shared data and set up the application state for rendering the current frame. This should be done in the application object method `latePreFrame()` or in `draw()`. This is demonstrated in Example 6.9, "Reading from Application-Specific Shared Data in `latePreFrame()`" and in Example 6.10, "Reading from Application-Specific Shared Data in `draw()`". All the nodes in the cluster will read the results of the computations made in `preFrame()` and set up the application state before rendering. In general, there should be no need to use `cluster::UserData<T>::isLocal()` at this point.

Example 6.9. Reading from Application-Specific Shared Data in `latePreFrame()`

```
void AppObject::latePreFrame()
{
   mStateVar1 = mMyTypeObj->getIntData();
   mStateVar2 = mMyTypeObj->getByteData();
   // And so on ...
}
```

Example 6.10. Reading from Application-Specific Shared Data in `draw()`

```
void AppObject::draw()
{
   int  state_var1 = mMyTypeObj->getIntData();
   char state_var2 = mMyTypeObj->getByteData();
   // And so on ...
   // Render the scene ...
}
```

The choice of which method to use depends on the application type and on the data flow of the application object. Scene graph-based application objects will not have a `draw()` method, so `latePreFrame()` must be used. For application object types not based on a scene graph (currently only `vrj::GlApp`), there is a trade off to consider. If `latePreFrame()` is used, then the rendering state information must be stored in member variables of the application class. If `draw()` is used, then the state can be defined using stack variables within the method, but the additional function call overhead and pointer derference (resulting from the use of the Smart Pointer pattern) could impact the application frame rate. Remember that `draw()` is invoked for every window and every viewport within each window. The number of calls to `draw()` increases further when stereoscopic rendering is enabled. If `latePreFrame()` is used instead, the Smart Pointer overhead will only be exhibited once per frame.

General Cluster Programming Issues

With the tools for VR Juggler cluster programming in hand, we can turn our attention to specific, higher level areas that must be handled carefully when writing applications that may run on a graphics cluster.

Time-Based Computations

Using time as input to algorithms is a very common occurrence in VR applications. On a cluster, however, each node has its own clock, and each node may start its frame loop at a slightly different time than the other cluster nodes. Differences such as these would result in inconsistencies among the time-based computations across the cluster nodes.

These problems can be avoided through a feature of the input data sharing feature of VR Juggler's cluster support. Every time a Gadgeteer device driver takes a sample from the input device, a time stamp is applied to the sample. This time stamp is included with the shared device data and can be accessed through the device interfaces used by the application objects. A time delta since the last frame can then be calculated. Use of this is demonstrated in Example 6.11, “Calculating Frame Deltas Using `vpr::Interval`”.

Example 6.11. Calculating Frame Deltas Using `vpr::Interval`

```
static vpr::Interval last_frame;
vpr::Interval current_frame = mHead->getTimeStamp();
vpr::Interval diff(current_frame - last_frame);
last_frame = current_frame; // You can get the delta in microseconds from
                            // vpr::Uint64 delta = diff.usecs();
```

Important

This technique implies that the Remote Input Manager plug-in (`RIMPlugin`) and the Start Barrier plug-in (`StartBarrierPlugin`) must be used by the Cluster Manager.

Random Numbers

Random numbers are, by definition, random. When two computers generate a random number, there is a high likelihood that they will generate different numbers. However, the algorithms used to generate random numbers on computers generate *pseudo-random* numbers. Pseudo-random numbers are generated by algorithms that have a predictable nature. Given a known starting point (called a *seed*), the sequence of numbers generated can be predicted. If the same algorithm is seeded identically on two separate computers, the two sequences of generated random numbers will be identical. Varying the seed allows the algorithms to generate different random sequences

This is a very important issue for VR application programming in a cluster configuration. When an application object uses random numbers, each application instance across the cluster must generate the same sequence of random numbers. With VR Juggler, there are two options for making this happen. The first is to seed the random number generator algorithm identically on all the cluster nodes. This is an easy solution as long as all the nodes use the same algorithm to generate random numbers. If the seed is hard coded into the application object initialization, the random number sequence will always be the same for every run of the application. While the numbers will still be random, the predictable nature of pseudo-random number generators could become a detriment.

The second option is to use application-specific shared data, as described above in the section called “Application-Specific Shared Data”. In this case, only one node will generate the random numbers, and the Application Data Manager will take care of sharing the most recently generated number(s) with the

other nodes. Using this approach allows for better algorithm seeding and thus better random number generation. It also avoids the issue of different computers having different random number generator algorithms.

Troubleshooting Cluster Problems

In this final section, we present some frequently asked questions regarding VR Juggler application programming and clustering.

1.1.
Why doesn't swap lock work with OpenGL Performer-based applications?

OpenGL Performer can make use of multiple processes to separate the App, Cull, and Draw actions. This allows Performer to spread its work out across three processors. Unfortunately, this interferes with cluster synchronization, so Performer multi-processing cannot be used in conjunction with the cluster capabilities in VR Juggler. The VR Juggler Performer Draw Manager is already written to disable multi-processing when the Cluster Manager is active.

1.2.
Why is my application navigating differently on every screen?

All navigation must be based on time stamps returned from input devices. These time stamps, of type `vpr::Interval`, can be acquired from any device interface that refers to a device being shared across the cluster, and they can be used to compute the time deltas between frames. More details can be found in the section called “Time-Based Computations”.

1.3.
Why does my application hang at startup on all the nodes?

When using the Start Barrier Plug-in, the application object frame loop methods (`preFrame()`, `postFrame()`, `draw()`, `getScene()`, etc.) will not be invoked until all the cluster nodes are ready to run. The “ready to run” determination is made by waiting for all the cluster nodes to be connected and for all the cluster plug-ins to complete their initialization. If all of the cluster nodes are running and connected, then there is probably a problem with a cluster plug-in.

For example, the Application Data Manager will not initialize correctly on all nodes if the type-specific GUIDs do not match on all nodes. In this case, disabling the Start Barrier Plug-in will result in the data-local cluster node being the only one that starts correctly. All the others will fail to open any display windows. See the section called “Application-Specific Shared Data” for more information on this topic.

Adding Audio

Immersive applications often take advantage of sound to enhance suspension of disbelief. VR Juggler application programmers have many options available to them for adding sound to their virtual environments. In this section, we describe the library Sonix that ships with VR Juggler. Sonix provides a layer of abstraction for third-party audio libraries including OpenAL [http://www.openal.org/], Audiere [http://audiere.sourceforge.net/], and AudioWorks [http://www.multigen-paradigm.com/products/runtime/vega/modules/audioworks.shtml]. In VR Juggler 2.0, there is one Sound Manager implemented using Sonix. As such, Sonix is the easiest tool to use for adding sound to VR Juggler applications, but it is not the only option.

Sonix provides simple audio sound objects on top of several audio APIs. The interface to Sonix is kept

very simple in order to get people up and running with sound as fast as possible. Sonix has the following features:

- Simple access to spatialized sound triggering.
- Small learning curve withsimple interface and usage.
- Abstracts several well-known audio systems to provide enhanced applicatin portability.
- Supports reconfigurability at runtime.
- Changing a sound resource reflects properly in all other handles to the same resource (resources allow multiple users).
- Reconfigurations of sound resources are protected (i.e. reconfiguration does not break application).
- Supports features needed by 3D virtual environments including spatialized audio, ambient audio, one-shot sounds, and looping sounds.

VR Juggler application programmers can use the full Sonix API directly, or they can take advantage of the VR Juggler Sound Manager to reduce the amount of coding required. We will present both approaches. Example applications can be found in the directory `$VJ_BASE_DIR/share/vrjuggler/samples/sound/simple`.

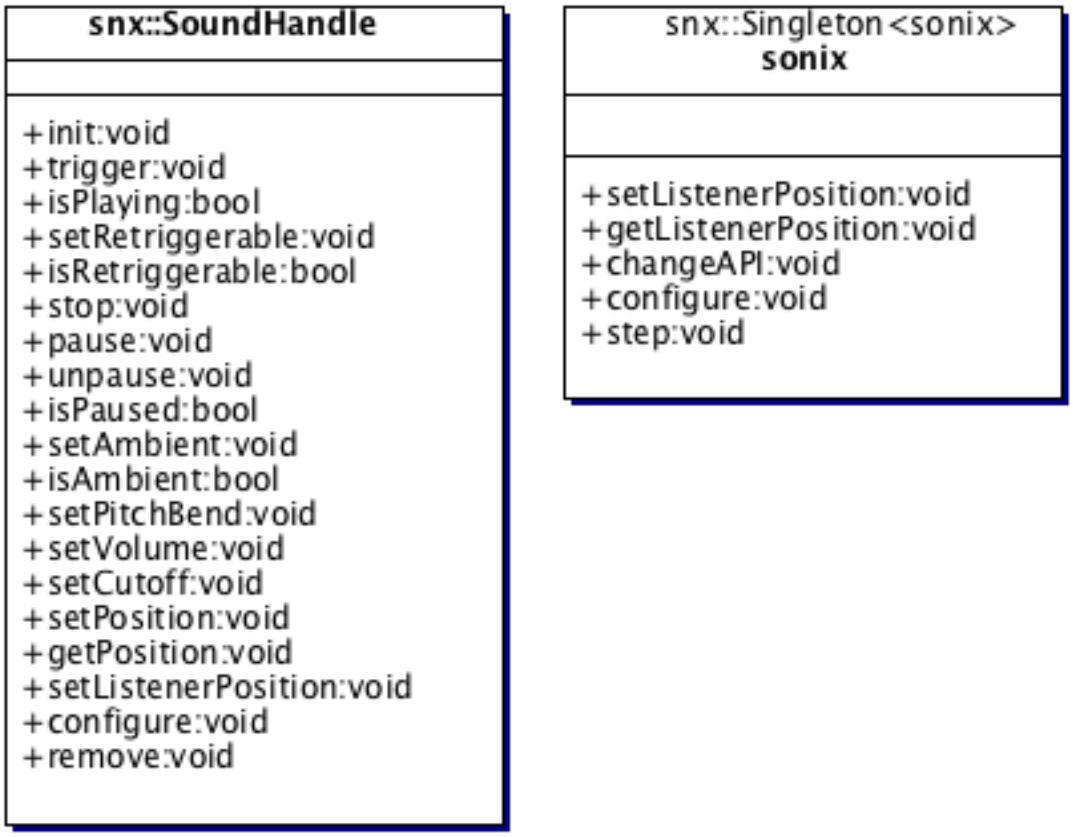
Using Sonix Directly

We begin our explanation of adding sound to a VR Juggler application by presenting the direct use of the Sonix API. This requires more programming than using the VR Juggler Sound Manager, but it will be useful to understand how Sonix operates in order to take full advantage of the Sound Manager. In particular, reading this section is a necessary part of understanding general Sonix usage, and readers are encouraged to read both this section and the next before deciding on which approach to use in their VR Juggler applications.

Basic Sonix Usage

In Figure 6.1, “Basic Sonix Interface”, we see the API for Sonix (see Figure 6.2, “The Sonix Design” for the complete software architecture). The main parts that a sound programmer will use are the classes `snx::SoundHandle` and `snx::sonix` (a singleton). The `snx::SoundHandle` class is used to manipulate individual sounds. The `snx::sonix` singleton class is used to start, stop, and reconfigure the sound system. Both classes must be used for any sound to be heard.

Figure 6.1. Basic Sonix Interface



Starting the Sonix system is easy. Basically, the only call needed to start the system is `snx::sonix::changeAPI()`. The backend audio software is loaded into Sonix through a plug-in system. The plug-in to use is identified by a string name, one of “OpenAL”, “AudioWorks”, or “Audiere”, as shown in Example 6.12, “Initializing the Sonix Sound APi”. The choice of which to use depends on availability and compatibility. For example, AudioWorks is available on the IRIX operating system, but it can only be used with a build of VR Juggler that uses SPROC threads. On the other hand, OpenAL and Audiere on IRIX can only be used with a pthreads build of VR Juggler.

Example 6.12. Initializing the Sonix Sound APi

```
snx::sonix::instance()->changeAPI("OpenAL");
```

Setting up a sound is designed to be simple, and an example of doing so is shown in Example 6.13, “Setting Up a Sonix Sound Handle”. Here, we use an instance of `snx::SoundInfo` to configure the sound object, which is in turn accessed by an instance of `snx::SoundHandle`.

Example 6.13. Setting Up a Sonix Sound Handle

```
snx::SoundInfo info;
info.filename = "crack.wav";
info.datasource = snx::SoundInfo::FILESYTEM;
```

```
snx::SoundHandle crack_sound("crack");
crack_sound.configure(info);
```

Periodically, an update function must be called to keep Sonix running. The method is snx::sonix::step(), and it takes a single argument of type float representing the time since the last time it was called. In general, this should be invoked in the frame function of an application. In the context of VR Juggler, this would be either `preFrame()` or `postFrame()` in the application object interface. An example of using this function is shown in Example 6.14, “Sonix Frame Update”.

Example 6.14. Sonix Frame Update

```
void MyApp::preFrame()
{
   // Update application state for the current frame
   // ...

   float time_delta = getTimeChangeInSeconds(); // use a system call or other API
                                                // to get the time delta
   snx::sonix::instance()->step(time_delta);
}
```

Example 6.15. Complete Sonix Program Using OpenAL

```
  1 #include <iostream>
    #include <string>
    #include <snx/sonix.h>

  5 int main(int argc, char* argv[])
    {
       std::string filename(:808kick.wav"), api("OpenAL");

       if ( ! snx::FileIO::fileExists(filename.c_str()) )
 10    {
          std::cout << "File not found: " << filename << "\n" << std::flush;
          return 1;
       }

 15    // start sonix using OpenAL
       snx::sonix::instance()->changeAPI(api);

       // fill out a description for the sound we want to play
       snx::SoundInfo sound_info;
 20    sound_info.filename = filename;
       sound_info.datasource = snx::SoundInfo::FILESYSTEM;

       // create the sound object
       snx::SoundHandle sound_handle;
 25    sound_handle.init("my simple sound");
       sound_handle.configure(sound_info);

       // trigger the sound
       sound_handle.trigger();
 30    sleep(1);

       // trigger the sound from a different position in 3D space
```

```
      sound_handle.setPosition(10.0f, 0.0f, 0.0f);
      sound_handle.trigger();
35    sleep(1);

      // this simulates a running application
      while ( 1 )
      {
40       snx::sonix::instance()->step(time_delta);
      }

      return 0;
   }
```

Reconfiguration

Sonix is reconfigurable, allowing audio APIs to be swapped out at run time safely without the dependent systems noticing. Applications using Sonix expect to be completely portable. Changing sound APIs at run time can be useful so that the user can experiment with quality and latency differences of different hardware and sound APIs. If no audio API is available on a given platform, calls to Sonix simply are ignored. This means that applications do not have to have special code to enable or disable sounds based on the availability of the backend sound software. These details are handled transparently by Sonix.

Everything in Sonix can be reconfigured behind the scenes during application execution. See Example 6.16, “Reconfiguring Sonix at Run Time” for a trivial example of how to reconfigure Sonix in C++. In a more complex context, this feature of Sonix has powerful implications. Sounds can be changed on the fly, as well as the API being used to render the sound. Such details are hidden behind sound handles and the `snx::sonix` singleton.

Example 6.16. Reconfiguring Sonix at Run Time

```
 1 // start sonix using OpenAL
   snx::sonix::instance()->changeAPI("OpenAL");

   // fill out a description for the sound we want to play
 5 snx::SoundInfo sound_info;
   sound_info.filename = "808kick.wav";
   sound_info.datasource = snx::SoundInfo::FILESYSTEM;

   // create the sound object
10 snx::SoundHandle sound_handle;
   sound_handle.init("my sound for testing");
   sound_handle.configure(sound_info);

   // trigger the sound
15 sound_handle.trigger();
   sleep(1);

   // trigger the sound using a different audio system
   snx::sonix::instance()->changeAPI("AudioWorks");
20 sound_handle.trigger();
   sleep(1);

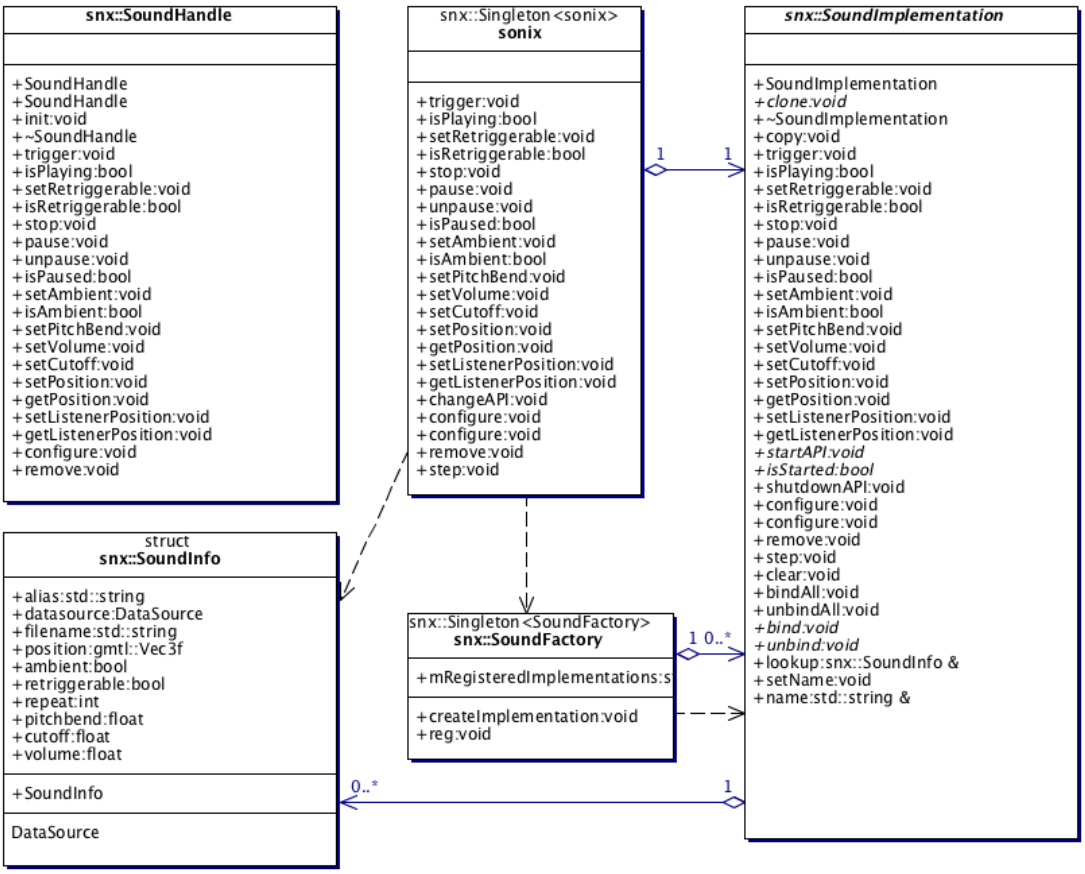
   // trigger our sound object using different source data
   sound_info.filename = "303riff.wav";
25 sound_handle.configure(sound_info);
   sound_handle.trigger();
   sleep(1);
```

Design and Implementation of Sonix

Before we conclude this section on using Sonix directly in applications, we examine the design details of Sonix. These details are beyond the scope of what needs to be understood in order for Sonix to be utilized by VR Juggler applications, so readers not interested in these details can skip ahead to the section called “Using the VR Juggler Sound Manager”.

Sonix was designed using modern software design principles including design patterns and object oriented design. The complete design of Sonix is depicted in UML in Figure 6.2, “The Sonix Design”. As evidenced by the diagram, the design of Sonix is quite simple, though the classes tend to have relatively large interfaces. Refer to the Sonix *Programmer Reference*, found on the Sonix documentaiton web page [http://www.vrjuggler.org/sonix/docs.php], for complete documentation of these interfaces.

Figure 6.2. The Sonix Design



Design Patterns Overview

Design patterns describe simple and elegant solutions to specific problems in object oriented software design. When designing Sonix, we used many design patterns appropriate for a simple audio system.

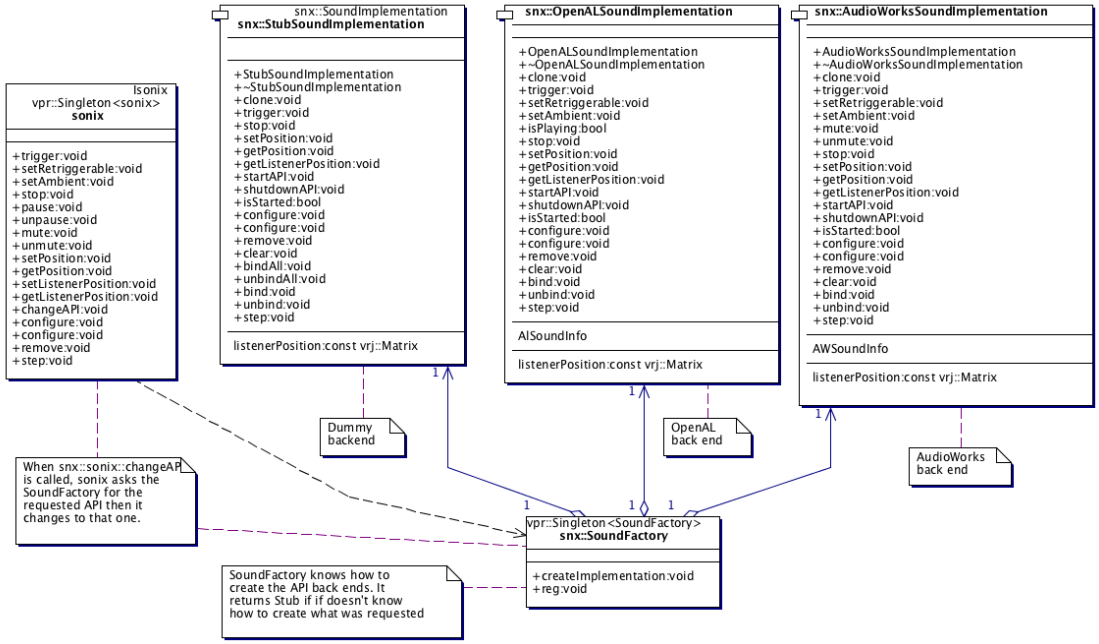
- Adapter (`snx::SoundImplementation`). This adapter provides a common interface to the underlying sound API.

- Prototype (`snx::SoundImplementation`). Making `snx::SoundImplementation` a Prototype allows a new cloned object to be created from it that has duplicate state.
- Store/plugin-method (`snx::SoundFactory`). Each sound implementation is registered with a Store called `snx::SoundFactory`. This Store allows users to select items from its inventory. Another name for Store is "Abstract Factory."
- Abstract Factory (`snx::SoundFactory`). The Store can create new instances of the requested sound implementation. The Abstract Factory consults its Store of registered objects, and if found, makes a clone of that object (Prototype pattern). The Abstract Factory is used in Sonix to configure the Bridge.
- Bridge (`snx::sonix` interface class and `snx::SoundImplementation`). The `snx::sonix` class is the audio system abstraction which is decoupled from its implementation `snx::SoundImplementation`. This way the two can vary independently. Bridge also facilitates run-time configuration of the sound API.
- Proxy (`std::string` and `snx::SoundHandle`). `snx::SoundHandle` is how users manipulate their sound object. `snx::SoundHandle` is actually a proxy to a `std::string` proxy. The `std::string` Proxy is what allows Sonix reconfiguration of resources. Rather than using pointers which can easily be left to dangle, the `std::string` serves as a lookup for a protected sound resource located internally to the Sonix run-time memory space. The `snx::SoundHandle` wraps this `std::string` to provide a simple and familiar C++ object to use as the sound handle. The Sonix class acts as Mediator between every Proxy method and the actual audio system Adapter.

Pluggable Audio Subsystems

Sonix supports the selection of several audio subsystems by the application through implementation plug-ins (see Figure 6.3, "Use of Plug-ins in Sonix"). Each plug-in implements an adapter to an underlying audio subsystem. The adapter supports a common interface that Sonix knows how talk to. Each adapter is then registered with a factory object, which may ask that adapter to clone itself for use by whomever called the factory.

Figure 6.3. Use of Plug-ins in Sonix



Using the VR Juggler Sound Manager

Now that we understand how Sonix works and how to use its simple API, we can go one step further and simplify the usage even more. Similar to the graphics API-specific VR Juggler Draw Managers, there exists the concept of a Sound Manager. There is a Sonix-specific Sound Manager implementation that handles many of the details of Sonix usage.

The Sonix Sound Manager is configured using a config element of type `sound_manager_sonix`. The config element sets the sound API to use, defines the llistener position, and sets up all the sound objects that will be used by the application. All of this would normally have to be done by the application programmer using the `snx::sonix` singleton, as presented above in the section called “Basic Sonix Usage”. In Example 6.17, “Example Sonix Sound Manager Conifguration”, we show a small example of configuring the Sonix Sound Manager. Note that some parts of the config file are removed for brevity. Aspects of this will be referenced in the code examples shown below.

Example 6.17. Example Sonix Sound Manager Conifguration

```
<sound_manager_sonix version="1">
   <api>OpenAL</api>
   <listener_position>0.0</listener_position>
   <listener_position>0.0</listener_position>
   <listener_position>0.0</listener_position>
   <sound>
      <sound name="bump" version="1">
         <filename>${VJ_BASE_DIR}/share/vrjuggler/data/sounds/bump.wav</filename>
         <ambient>false</ambient>
         <retriggerable>false</retriggerable>
         <loop>1</loop>
         <pitch_bend>1.0</pitch_bend>
         <cutoff>1.0</cutoff>
         <volume>1.0</volume>
         <position>0.0</position>
         <position>0.0</position>
```

```
         <position>0.0</position>
      </sound>
   </sound>
   <sound>
      <sound name="step" version="1">
         <filename>${VJ_BASE_DIR}/share/vrjuggler/data/sounds/footstep.wav</filena
         <ambient>false</ambient>
         <retriggerable>false</retriggerable>
         <loop>1</loop>
         <pitch_bend>1.0</pitch_bend>
         <cutoff>1.0</cutoff>
         <volume>1.0</volume>
         <position>0.0</position>
         <position>0.0</position>
         <position>0.0</position>
      </sound>
   </sound>
   <!-- Other sound objects ... -->
</sound_manager_sonix>
```

All sound files named in the config element are automatically loaded by the Sonix Sound Manager. User applications simply declare sound handles that refer to the loaded sound objects. The VR Juggler application class will then include one `snx::SoundHandle` instance for each sound object declared in the Sonix Sound Manager config element, as shown in Example 6.18, “Declaring Sound Handles in Application Object Class”. This is very similar to the use of Gadgeteer device interfaces that provide access to device input.

Example 6.18. Declaring Sound Handles in Application Object Class

```
class MySoundApp : public vrj::GlApp
{
public:
   void init();
   void preFrame();
   // Other application object interface methods ...

private:
   // Determines if the "bump" sound should be triggered.
   bool shouldTriggerBump();

   // Determines if the "step" sound should be triggered.
   bool shouldTriggerStep();

   snx::SoundHandle mBumpSound;
   snx::SoundHandle mStepSound;
   // And so on ...
};
```

To get access to the sounds through the `snx::SoundHandle` objects, the sound handles must be initialized. This is done in the same way as described earlier in the explanation of direct use of the Sonix API. The sound handle initialization should be performed in the `init()` method of the VR Juggler application object, as shown in Example 6.19, “Initializing Sound Handles in an Application Object”. The string value passed to `snx::SoundHandle::init()` is the name given in the Sonix Sound Manager config element. Again, we see similarity with the usage of Gadgeteer device interfaces.

Example 6.19. Initializing Sound Handles in an Application Object

```
void MySoundApp::init()
{
   mBumpSound.init("bump");
   mStepSound.init("step");
   // And so on ...
}
```

Finally, the sounds can be triggered in the application object `preFrame()` method, shown below in Example 6.20, “Triggering Sounds in an Application Object”. They should not be triggered in the `draw()` method because, as we have seen in earlier sections, `draw()` can be invoked multiple times per frame. There is no need to call the function `snx::sonix::step()` in `preFrame()` or in `postFrame()`. This is done automatically within the Sonix Sound Manager each time through the kernel control loop.

Example 6.20. Triggering Sounds in an Application Object

```
void MySoundApp::preFrame()
{
   if ( shouldTriggerBump() )
   {
      mBumpSound.trigger();
   }

   if ( shouldTriggerStep() )
   {
      mStepSound.trigger();
   }
}
```

Other methods in the `snx::SoundHandle` class interface can be invoked on the sound handle objects. We have focused on simple rendering of sounds here. Refer to the Sonix *Programmer Reference*, found on the Sonix documentation web page [http://www.vrjuggler.org/sonix/docs.php], for complete documentation on the interface of `snx::SoundHandle`.

Chapter 7. Porting to VR Juggler from the CAVElibs™

In this chapter, we give some methods for porting an application written with the CAVElibs™ 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 CAVElibs™ software, and we translate concepts familiar to CAVElibs™ programmers into VR Juggler terms.

The Initialize, Draw, and Frame Routines

In the CAVElibs™, 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.

In CAVElibs™

The following lists the draw, frame, and initialize routines used in the CAVElibs™ 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()`

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 2, *Application Basics*. Briefly, the application class definition would appear similar to the following:

```
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 the section called “OpenGL Applications”. For information about context-specific data, refer to the section called “Context-Specific Data”.

Getting Input from Devices

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

In CAVElibs™

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()`

In VR Juggler

To get device input, use the type-specific instantiations of `gadget::DeviceInterface<T>`. 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
- `gadget::KeyboardMouseInterface` for keyboard and mouse input

For more information about the VR Juggler device interfaces, refer to the section called “Device Proxies and Device Interfaces”. A tutorial on getting device input in VR Juggler applications can be found in the section called “Getting Input”.

Configuration

Configuration of VR Juggler and the CAVElibs™ 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.

In CAVElibs™

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.

In VR Juggler

Configuration of VR Juggler is much more powerful and flexible than what is used by the CAVElibs™ 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.

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.

Shared Memory

Unlike the CAVElibs™ 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.

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 CAVElibs™ 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 the section called “Context-Specific Data”.

Source Code

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

The Form of a Basic CAVElibs™ Program

```
 1 void app_shared_init();
   void app_compute_init();
   void app_init_gl();
   void app_draw();
 5 void app_compute();

   void main(int argc, char **argv)
   {
      CAVEConfigure(&argc,argv,NULL);
10    app_shared_init(argc,argv);
      CAVEInit();
      CAVEInitApplication(app_init_gl,0);
      CAVEDisplay(app_draw,0);
      app_compute_init(argc,argv);
15    while (!getbutton(ESCKEY))
      {
          app_compute();
      }
      CAVEExit();
20 }
```

The Form of a Basic VR Juggler Program

```
 1 class MyApplication : public vrj::GlApp
   {
   public:
   // Data callbacks  (Do not put OpenGL code here)
 5    virtual void init();
      virtual void preFrame();
      virtual void intraFrame();
      virtual void postFrame();

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

15 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++)
20    {
         // loading config file passed on command line...
         kernel->loadConfigFile(argv[i]);
      }

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

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

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

35    return 0;
   }
```

Chapter 8. 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.

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.

The Initialize, Draw, and Frame Routines

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 `glutDisplay-Func()` (once for each window opened).

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 2, *Application Basics*. Briefly, the application class definition would appear similar to the following:

```
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 the section called “OpenGL Applications”. For information about context-specific data, refer to the section called “Context-Specific Data”.

Getting Input from Devices

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)`

In VR Juggler

To get device input, use the type-specific instantiations of `gadget::DeviceInterface<T>`. 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
- `gadget::KeyboardMouseInterface` for keyboard and mouse input

For more information about the VR Juggler device interfaces, refer to the section called “Device Proxies and Device Interfaces”. A tutorial on getting device input in VR Juggler applications can be found in the section called “Getting Input”.

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.

In GLUT

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

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 VR Juggler *Configuration Guide* for more information on configuring VR Juggler.

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.

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()`.

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 a more detailed description of these concepts and a tutorial on how to use them, please refer to the section called “Context-Specific Data”.

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.

The Form of a Basic GLUT Program

```
 1 void main(int argc, char* argv[])
   {
      /* initialize the application data here */
      OnApplicationInit();
 5
      /* create a window to render graphics in
       * In VR Juggler, window creation is done for you based on your configurati
       * settings.
       */
10    glutInitWindowSize( 640, 480 );
      glutInit( &argc, argv );
      glutInitDisplayMode( GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE );
      glutCreateWindow( "GLUT application" );

15    /* 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)
20     */
      glutReshapeFunc( OnReshape );
      glutIdleFunc( OnIdle );
      glutDisplayFunc( OnIdle );

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

30    /* keyboard callback functions. */
      glutKeyboardFunc( OnKeyboardDown );
      glutKeyboardUpFunc( OnKeyboardUp );
      glutSpecialFunc( OnSpecialKeyboardDown );
      glutSpecialUpFunc( OnSpecialKeyboardUp );
35
      /* mouse callback functions... */
      glutMouseFunc( OnMouseClick );
      glutMotionFunc( OnMousePos );
      glutPassiveMotionFunc( OnMousePos );
40
      /* 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
       * (see below)
       */
```

```
45    glutMainLoop();
   }
```

The Form of a Basic VR Juggler Program

```
 1 class MyApplication : public vrj::GlApp
   {
   public:
   // Data callbacks  (Do not put OpenGL code here)
 5    virtual void init();
      virtual void preFrame();
      virtual void intraFrame();
      virtual void postFrame();

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

15 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++)
20    {
         // loading config file passed on command line...
         kernel->loadConfigFile(argv[i]);
      }

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

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

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

35    return 0;
   }
```

Part III. Advanced Topics

Table of Contents

Chapter 9. System Interaction

In this part of the book, we present information for advanced users who want to create applications that take advantage of VR Juggler features such as threading and run-time reconfiguration. While we do recommend that all programmers be familiar with these topics, readers who are not familiar with the basic concepts of multi-threaded programming, for example, may find these chapters difficult to understand.

Chapter 10. Multi-threading

In this chapter, we present how to use multi-threading within VR Juggler applications. Readers who are not familiar with the basic concepts of multi-threaded programming may find the following sections difficult to understand. This chapter is written with the assumption that readers already know the necessary background material and want to learn about how VR Juggler implements the concepts.

Techniques

VR Juggler is a multi-threaded software system. We have built up a cross-platform abstraction for threads and synchronization primitives as part of making VR Juggler more portable. This abstraction is available to application developers. In addition, the basic VR Juggler application object interface provides a mechanism for inherent parallel programming in applications. In this section, we provide a more detailed description of these techniques and how to put them into use.

To begin the discussion on multi-threaded programming with VR Juggler, we describe the techniques available to application programmers. There are three options from which programmers may choose:

1. Use the `vrj::App::intraFrame()` application object member function
2. Use triple-buffered data with `vrj::App::intraFrame()`
3. Use triple-buffered data with separate threads

Tutorial: Perform Computations Asynchronously to Rendering with `intraFrame()`

In this section, we present a tutorial demonstrating the use of asynchronous processing and rendering. The tutorial overview is as follows:

- Description: Performing the computations to animate a mesh asynchronously to rendering the mesh.
- Objectives: Understand how to use multi-threading techniques in an application.
- Member functions: `vrj::App::init()`, `vrj::App::preFrame()`, `vrj::App::intraFrame()`, `vrj::GlApp::draw()`
- Directory: `$VJ_BASE_DIR/share/samples/OGL/simple/MPApp`
- Files: `MPApp.h`, `MPApp.cpp`, `Mesh.h`

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 the section called “Tutorial: Drawing a Cube with OpenGL”. Note the addition of the `contextInit()` declaration and the use of the context-specific data member `mCubeDlId`.

```
1 using namespace vrj;
```

```
class MPApp : public GlApp
{
public:
   MPApp() {;}
   virtual void init();
   virtual void preFrame();
   virtual void draw();
   virtual void intraFrame();
    ...
public:
   // The Mesh object used for computing and rendering.
   Mesh mMesh;
   ...
};
```

The `preFrame()` Member Function

We now show the implementation of `MPApp::preFrame()`. In the initial version of the application, the computation for the mesh animation is done here.

```
void MPApp::preFrame()
{
   mCurTime += 0.005;
   mMesh.compute(mCurTime);

   // Other once-per-frame operations follow ...
}
```

Note

Note that in this example, the animation is based on the frame rate. It is much better to use a time-based animation scheme. For this example, however, we will keep things simple.

The `draw()` Member Function

When we enter `draw()`, the computations for this frame's animation are complete. Thus, we can call the `render()` method to display the new calculations.

```
using namespace gmtl;

void MPApp::draw()
{
   glClear(GL_DEPTH_BUFFER_BIT);

   // --- Setup for drawing --- //
   glMatrixMode(GL_MODELVIEW);

   glColor3f(0.0f, 0.5f, 0.75f);
   glPushMatrix();
      glScalef(15.0f,15.0f,15.0f);
      glRotatef(-90.0f,1.0,0.0,0.0);
      mMesh.render();                // Render the mesh
   glPopMatrix();
}
```

Exercise

In the tutorial application code, perform the computations asynchronously to the rendering process by using `intraFrame()`. Recall from the section called "`vrj::App::intraFrame()`" (in the section called "Frame Functions") that `intraFrame()` is invoked in parallel to the execution of `draw()`.

Helper Classes

All the techniques presented in the previous section require some form of synchronization to protect the data accessed by multiple threads. Developers who choose the third option and use separate threads must learn the VR Juggler Portable Runtime (VPR) thread API. A full description of this API is beyond the scope of this book. In this section, we will describe at a high level the important classes. For more detailed information, refer to the *VR Juggler Portable Runtime Programmer's Guide*.

`vpr::Thread`

When considering multi-threaded programming, it is important to know that with great power comes great responsibility. The power is being able to provide multiple threads of control in a single application. The responsibility is making sure those threads get along with each other and do not step on each other's data. VR Juggler is a multi-threaded library which makes it very powerful and very complex.

As a cross-platform framework, VR Juggler uses an internal threading abstraction that provides a uniform interface to platform-specific threading implementations. That cross-platform interface is available to programmers to make applications multi-threaded without tying them to a specific operating system's threading implementation.

The threading interface in VPR is modeled after the POSIX thread specification of POSIX.1b (formerly POSIX.4). The main difference is that VPR's interface is object-oriented while POSIX threads (pthreads) are procedural. The basic principles are exactly the same, however. A function (or class method) is provided to the `vpr::Thread` class, and that function is executed in a thread of control that is independent of the creating thread.

Threads are spawned (initialized and begin execution) when the `vpr::Thread` constructor is called. That is, when instantiating a `vpr::Thread` object, a new thread of execution is created. The semantics of threads says that a thread can begin execution at any time after being created, and this is true with `vpr::Thread`s. Do not make any assumptions about when the thread will begin running. It may happen before or after the constructor returns the `vpr::Thread` object.

To pass arguments to threads, the common mechanism of encapsulating them in a C++ struct must be used. The function executed by the thread takes only a single argument of type void*. An argument is not required, of course, but to pass more than one argument to a thread, the best way to do this is to create a structure and pass a pointer to it to the `vpr::Thread` constructor.

Once a `vpr::Thread` object is created, it acts as an interface into controlling the thread it encapsulates. Thread signals can be sent, priority changes can be made, execution can be suspended, etc. This interface is the focus of this section.

`vpr::BaseThreadFunctor`

In this section, we explain the concept and use of functors. As with much of VR Juggler, a functor is a high-level concept that encapsulates something quite simple. A functor is defined as "something that performs an operation or a function." While this is not very detailed, it is clear and concise. In VPR, functors can be used as the code executed by a thread.

In VPR, a functor is simply another object type that happens to encapsulate a user-defined function. The details on how this is done are not important here, but they are provided later for those who are interested. What is important to know is that a functor can be thought of as a normal function. When using

them, programmers simply implement a function and then pass the function pointer (and the function's optional argument) to the functor's constructor. The object does the rest.

Observant readers may have noticed the parenthetical phrase in the previous paragraph mentioning a function's optional argument. Note that “argument” is singular meaning that only one parameter can be passed to the function that will be run by the created thread. The type of that argument is the wonderfully vague void*, an artifact of basing the threading subsystem on C libraries. As discussed in the section on using `vpr::Thread`s, if there is a need to pass multiple arguments, they must be encapsulated in a struct or a comparable object.

Once a functor object exists, it is passed to the `vpr::Thread` constructor, and the new thread will execute the functor (which knows about the function). The end result is the same as using a normal C/C++ function or a static class member function, but there is one special benefit: with functors, non-static class member functions can be passed. In many cases, there arises a need to run a member function in a separate thread, but making it static is infeasible or awkward. Thus, it would be best to pass a non-static member function to the created thread. To get access to the non-static data members, however, the C++ `this` pointer must be available to the thread. By using a VPR functor, that is all handled behind the scenes so that passing a non-static member function is straightforward.

Using the `vpr::Semaphore` Interface

The most important part of multi-threaded programming is proper thread synchronization so that access to shared data is controlled. Doing so results in consistency among all threads. Semaphores are a very common synchronization mechanism and have been used widely in concurrent systems. This short section describes the cross-platform semaphore interface provided with and used by VPR. It does not explain what semaphores are or how to use them—it is assumed that readers are already familiar with the topic lest they probably would not be reading this chapter on advanced classes at all.

As with threads, a cross-platform abstraction layer has been written into VPR to provide a consistent way to use semaphores on all supported platforms. The primary goal behind the interface design is to provide the common *P* (acquire) and *V* (release) operations. A given semaphore controls access to *n* resources. Thus, n threads can acquire a semaphore without blocking, but the $n + 1$ attempt to acquire the semaphore will block until a resource release occurs. The interface does include methods for read/write semaphores, but as of this writing, that part of the interface is not complete.

Using the `vpr::Mutex` Interface

In addition to cross-platform semaphores, VPR provides an abstraction for cross-platform mutexes. Mutexes are a special type of semaphore known as a binary semaphore. Exactly one thread can hold the lock at any time. This very short section, however, is not about mutexes but rather about the `vpr::Mutex` interface provided with VPR and used by VR Juggler.

The cross-platform mutex abstraction in VPR is critical for synchronizing access to shared data. Those who have read the section on `vpr::Semaphore` will find this section very, very familiar. The interface for `vpr::Mutex` is a subset of that for `vpr::Semaphore` since mutexes are binary semaphores. In other words, they control access to exactly one resource. They can be locked and unlocked. That is all there is to know. The `vpr::Mutex` interface does include some methods for read/write mutexes, but this implementation is incomplete and is not documented here for that reason. When the implementation is finished, this documentation will be expanded.

Using Data Buffering

In the reader exercise from the section called “Tutorial: Perform Computations Asynchronously to Rendering with `intraFrame()`”, we do not suggest the use of mutexes to protect access to the data modified in `intraFrame()` and rendered in `draw()`. When the mesh is rendered, some tearing will almost certainly be seen. This happens as a result of data changing during the rendering of the data. In oth-

er words, access to a *critical section* of code not being controlled. To prevent tearing, we need to protect the internal mesh data. However, using a mutex to control access to the mesh will result in the computation and rendering being serialized. In so doing, we lose the value of parallelizing the computation and rendering.

Triple Buffering

To work around this, we can use multiple buffers. More specifically, we will use three buffers: one for reading, one for writing, and one as temporary storage of the most recently completed write(s). This is called *triple buffering*, and it allows us to perform computation and rendering in parallel without causing one process to wait on the other. The rendering buffer can be accessed without locking, as can the computation buffer. The only time a lock must be held is when the latest computations are copied into the temporary storage buffer or when the contents of the temporary storage buffer are copied into the rendering buffer. Thus, the key to triple buffering is the temporary storage buffer.

Implementing triple buffering is relatively straightforward, once the basic idea is understood. For this discussion we will refer to the buffers as follows:

1. *Stable*: The buffer from which the most recent data will be read
2. *Working*: The buffer into which the current data will be written
3. *Working Copy*: The buffer used to store the last completed write results temporarily

Of the three, Working Copy is the only one that is shared between the threads, and thus, only one mutex must be used. The mutex must be locked when swapping the contents of Stable and Working Copy and when swapping the contents of Working and Working Copy. No lock must be held when reading from Stable or when writing to Working. This is where the major benefit of triple buffering is realized.

In the thread that will write to Working, the implementation will follow this basic structure:

```
while ( KeepWriting )
{
   Write to Working

   Lock Working Copy mutex
      Swap Working and Working Copy
   Unlock Working Copy mutex
}
```

In the thread that will read from Stable, the implementation is (roughly) the reverse:

```
while ( KeepReading )
{
   Lock Working Copy mutex
      Swap Stable and Working Copy
   Unlock Working Copy mutex

   Read from Stable
}
```

Optimizing Triple Buffering

It is possible to optimize triple buffering to reduce the amount of time spent in the locked section. Instead of swapping the buffers, we can swap the pointers to the buffers. The result is that we only have to swap (up to) eight bytes of memory, regardless of the size of the actual data.

A very simple way to implement this is to use a three-element array to hold the buffers and three variables to use as the indices into the array. The variables can be named `stable`, `working`, and `working_copy`, for example. All accesses into the array of buffers use one of those three values. In other words, the thread reading from Stable will use `buffer[stable]` to get its buffer. Similarly, the thread writing to Working will use `buffer[working]`. Swapping the buffers simply involves changing the values of `stable` and `working_copy` or `working` and `working_copy`.

In the thread that will perform the writing, the pseudo-code for swapping is as follows:

```
Lock Working Copy mutex
   temp = working
   working = working_copy
   working_copy = working
Unlock Working Copy Mutex
```

Similarly, in the thread that will perform the reading, the pseudo-code for swapping is as follows:

```
Lock Working Copy mutex
   temp = stable
   stable = working_copy
   working_copy = stable
Unlock Working Copy Mutex
```

In these examples, it takes three memory copies to swap the two values. Use of the temporary storage is required (at the C/C++ level), and thus, swapping the memory cannot be done any faster, regardless of the size of the data buffers.

Using Triple Buffering in an Application

Triple buffering can be used in VR Juggler applications in two ways: in `intraFrame()` or in a separate thread. In both cases, the operations will happen asynchronously to the `draw()` thread, so the uses are *roughly* equivalent. Using `intraFrame()` is easier than creating a separate thread because the parallel processing comes "for free" with `intraFrame()`. No special actions must be taken on the part of the application developer. Programmers must keep in mind, however, that if the computations will take longer than the rendering, the application frame rate will drop. This is because the next frame cannot start until after `intraFrame()` completes. Thus, using a thread will increase the complexity of the application code somewhat, but it will ensure that the frame rate will not be adversely affected.

Tutorial: Perform Computations Using Triple Buffering

In this tutorial, we will use MPApp again to do parallel computation and rendering. This time, however, we will use triple-buffered, protected data instead of simply writing in `intraFrame()` and reading (simultaneously) in `draw()`. It will be best to begin with an unmodified version of MPApp to ensure that no changes made for the previous tutorial affect this exercise.

Using the information presented earlier in this section, the exercise is to extend the `Mesh` class (found in `Mesh.h`) so that it uses triple buffering for computation and rendering. It is up to the reader to decide whether to use threads (refer to the documentation for `vpr::Thread` in the section called "`vpr::Thread`" and in the online *Programmer Reference*) or to use `intraFrame()`. Either way, a single `vpr::Mutex` instance will be needed, so it may be helpful to review the section called "Using the `vpr::Mutex` Interface" as well.

Chapter 11. Run-Time Reconfiguration

In this chapter, we introduce run-time reconfiguration, one of the most powerful features of VR Juggler. We will give an overview of how it works before proceeding into how to use it. The idea here is to introduce the concepts, justify the value of run-time reconfiguration, and then present its use so that developers can take full advantage of this feature.

How Run-Time Reconfiguration Works

Reasons to Use Run-Time Reconfiguration

Using Run-Time Reconfiguration in an Application

There are four steps involved in adding run-time reconfiguration to a VR Juggler application. We describe each of them in detail here in the following subsections.

Create Application-Specific Configuration Definitions

The first step in adding dynamic reconfiguration capabilities to an application is to decide what aspects of the application should be configurable. Naturally, this is very application-specific, but some of the following choices are common:

- Initial parameters (position, color, etc.) of objects in the environment
- Navigational position
- Global settings such as difficulty level of a game, or network settings for a distributed application

Once decisions are made regarding configuration information, it is time to define the kinds of configuration elements that will contain it. This essentially means creating a file containing one or more configuration definitions. To understand this better, consider the following example. One might define an "Object" definition in an application. The definition would have properties that include the name and type of an object, its color and size, and so forth.

There are several ways to ensure that custom configuration definitions are read by the application. One way is to load the definition file explicitly (described below), but the simplest way is to include the custom definition file from one of the configuration files the application loads at startup. Instructions for editing configuration definition files and creating new kinds of definitions are included in the *VRJConfig User's Guide*.

Implement the Dynamic Reconfiguration Interface

The next step is to implement the dynamic reconfiguration interface for the application object. This interface is defined by the `jccl::ConfigElementHandler` class and consists of three methods:

1.

```
virtual bool configCanHandle(jccl::ConfigElementPtr element);
```

This function should simply return a Boolean (true or false) depending on whether this object knows how to deal with the configuration element passed to it. If this element uses an application-custom definition, this should return `true`. For example:

```
std::string s = element->getType();

if ( !jccl::strcasecmp(s, "my_custom_element_type") )
{
   return true;
}
```

2.
```
virtual bool configAdd(jccl::ConfigElementPtr element);
```

This method is called whenever an element is added to the application, whether by loading a configuration file or through a dynamic reconfiguration event. Prior to this, the element will have been passed through the application object's `configCanHandle()` method. Thus, when `configAdd()` is called, the element is destined for the application object.

When `configAdd()` is called, the application should look at the element passed to it and decide what to do. This might involve creating a new object, changing the configuration of an extant object, changing the values of certain variables, or any number of other possibilities. This flexibility is part of the power of dynamic reconfiguration with VR Juggler.

3.
```
virtual bool configRemove(jccl::ConfigElementPtr element);
```

This method is analogous to `configAdd()`. It is called when VR Juggler receives a command to remove a particular configuration element. If the element refers to a specific object in the application, the most obvious behavior would be to remove that particular object. If the element refers to some other properties of the application, there are several choices for the correct behavior. For example, one might choose to reset those properties to their default values. In some cases, it may be desirable or necessary to ignore the remove request.

Procesing Configuration Elements

When an application receives a configuration element to process via `configAdd()` or `configRemove()`, it needs to retrieve the data in that element in order to decide what to do. Configuration elements can be very complex, but the interface has been designed to be as simple as possible. We now describe a few of the most important methods in the `jccl::ConfigElement` API.

•
```
std::string jccl::ConfigElement::getID() const;
```

This method returns the token of the configuration definition which describes this element. This is useful if an application uses several kinds of custom element types. With this method, it is possible to distinguish one from another.

•
```
T jccl::ConfigElement::getProperty<T>(std::string& propertyToken,
```

```
                                        int num);
```

This is the key method for getting the information contained in a configuration element. Its arguments are the token associated with a property and a numeric index. For example, a property might store a coordinate with three values, each of which can be accessed separately by using the numbers 0, 1, or 2 for the `num` parameter.

The return value and the template parameter for this method need some explanation. Basically, the return value of `getProperty()` is cast to whichever type the caller expects to receive. This type is named using the template paramter. (It is assumed that the caller knows the types of values stored in a given property.) The use of the template paramter allows any type to be used through the type-specific specialization performed by the C++ compiler. The following code fragment gives a few examples of this usage:

```
std::string s1 = element->getProperty<std::string>("name", 0);

bool b = element->getProperty<bool>("enabled");
std::string s3 = element->getProperty<std::string>("enabled");
// s3 will be one of the strings "0" or "1"

jccl::ConfigElementPtr elt = element->getProperty<jccl::ConfigElementPtr>("child_
```

- ```
  int jccl::ConfigElement::getNum(const std::string& propertyToken) const;
  ```

  Sometimes properties of a configuration element can have a variable number of values. A good example is a property that lists a set of files to be loaded. The `getNum()` method returns the actual number of values of the named property.

For definitive information about the `jccl::ConfigElement` API, refer to the JCCL *Programmer's Reference*.

Loading and Saving Configurations

Tutorial: Using Application-Specific Configurations

In this section, we present a tutorial covering the use of application-specific configurations. The tutorial overview is as follows:

- Description: Simple OpenGL application that draws a cube in the environment.
- Objectives: Understand how the `configCanHandle()` and `configAdd()` member functions in `jccl::ConfigElementHandler` work; create application-specific configurations using JCCL
- Member functions: `jccl::ConfigElementHandler::configCanHandle()`, `jccl::ConfigElementHandler::configAdd()`, `vrj::GlApp::draw()`
- Directory: `$VJ_BASE_DIR/share/samples/OGL/simple/ConfigApp`
  ```

- Files: `config_app.jdef`, `ConfigApp.jconf`, `ConfigApp.h`, `ConfigApp.cpp`

Class Declaration

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

```
 1 using namespace vrj;
   using namespace gadget;

   class simpleApp : public GlApp
 5 {
   public:
      simpleApp();
      virtual bool configCanHandle(jccl::ConfigElementPtr element);
      virtual bool configAdd(jccl::ConfigElementPtr element);
10    virtual void init();
      virtual void draw();

   public:
      PositionInterface mWand;
15    DigitalInterface mButtons[3];

      GLfloat mSurfaceColor_ll[3];
      GLfloat mSurfaceColor_lr[3];
      GLfloat mSurfaceColor_ur[3];
20    GLfloat mSurfaceColor_ul[3];
   };
```

Application Configuration

Note

Readers unfamiliar with the JCCL terminology should refer to the JCCL documentation. We will not attempt to give a full explanation of how JCCL works here.

In the example application ConfigApp, we will see how to configure the color of the ground in the 3D scene using JCCL. The configuration is based on a configuration element definition that defines four properties, each with three values. The ground is defined as a rectangular polygon, and each property corresponds to one of the corners of the rectangle. The values of the property are the red, green, and blue values for the color of the corresponding corner vertex.

In Example 11.1, “Complete listing of `config_app.jdef`”, we show all of `config_app.jdef`. This contains many key pieces of information. The “name” attribute defines the type of the configuration element(s) that will appear in a configuration file. In our case, our config element type is “config_app.” Under the `definition_version` tag, the category for our configuration element is defined to be “Application.” This is done to indicate that this is an application-specific configuration definition. Finally, we see the property description for the corner colors, each of which has three floating-point values. The values have defaults of 1.0 for red, 0.0 for green, and 0.0 for blue. As such, the lower left corner vertex color defaults to red. The property definitions for the other three corners are similar.

Example 11.1. Complete listing of `config_app.jdef`

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings definition.version="3.1"?>
<definition xmlns="http://www.vrjuggler.org/jccl/xsd/3.1/definition"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.1/definition h
            name="config_app">
   <definition_version version="1" label="ConfigApp configuration">
      <help>This is an application-specific configuration definition for use with
      <parent/>
      <category>/Application</category>
      <property valuetype="float" variable="false" name="lower_left_color">
         <help>Specifies the RGB values for the lower left corner color.</help>
         <value label="Red value" defaultvalue="1.0"/>
         <value label="Green value" defaultvalue="0.0"/>
         <value label="Blue value" defaultvalue="0.0"/>
      </property>
      <property valuetype="float" variable="false" name="lower_right_color">
         <help>Specifies the RGB values for the lower right corner color.</help>
         <value label="Red value" defaultvalue="1.0"/>
         <value label="Green value" defaultvalue="0.0"/>
         <value label="Blue value" defaultvalue="0.0"/>
      </property>
      <property valuetype="float" variable="false" name="upper_right_color">
         <help>Specifies the RGB values for the upper right corner color.</help>
         <value label="Red value" defaultvalue="1.0"/>
         <value label="Green value" defaultvalue="0.0 "/>
         <value label="Blue value" defaultvalue="0.0"/>
      </property>
      <property valuetype="float" variable="false" name="upper_left_color">
         <help>Specifies the RGB values for the upper left corner color.</help>
         <value label="Red value" defaultvalue="1.0"/>
         <value label="Green value" defaultvalue="0.0 "/>
         <value label="Blue value" defaultvalue="0.0"/>
      </property>
      <upgrade_transform/>
   </definition_version>
</definition>
```

Using this configuration definition, we can create an instance of our configuration type in a configuration file. An example instance is shown in Example 11.2, “`ConfigApp.jconf`”. Here, we see the configuration for all four corners of the rectangular ground polygon. Note also that no special steps have to be taken in the configuration file to load the definition file. This is handled automatically by the JCCL Configuration Manager as long as the environment variable `$JCCL_DEFINITION_PATH` includes the directory where `config_app.jdef` sits. A typical way of setting this environment variable for this application would be the following:

```
% JCCL_DEFINITION_PATH = $VJ_BASE_DIR/share/data/definitions:$VJ_BASE_DIR/share/sa
```

Tip

If the syntax above for setting an environment variable does not seem familiar, refer to the chapter titled “Environment Variables” in the VR Juggler *Getting Started Guide* for a description of this shell-agnostic form.

Caution

It is critical that $VJ_BASE_DIR/share/data/definitions be included or else all the “core” `.jdef` files will not be found by the Configuration Manager at run time.

Example 11.2. `ConfigApp.jconf`

```
<?xml version="1.0" encoding="UTF-8"?>
<?org-vrjuggler-jccl-settings configuration.version="3.0"?>
<configuration xmlns="http://www.vrjuggler.org/jccl/xsd/3.0/configuration"
               name="ConfigApp Example Configuration"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://www.vrjuggler.org/jccl/xsd/3.0/configura
   <elements>
      <config_app name="Config App Example" version="1">
         <lower_left_color>1.0</lower_left_color>
         <lower_left_color>0.0</lower_left_color>
         <lower_left_color>0.0</lower_left_color>
         <lower_right_color>1.0</lower_right_color>
         <lower_right_color>1.0</lower_right_color>
         <lower_right_color>0.0</lower_right_color>
         <upper_right_color>0.0</upper_right_color>
         <upper_right_color>1.0</upper_right_color>
         <upper_right_color>0.0</upper_right_color>
         <upper_left_color>0.0</upper_left_color>
         <upper_left_color>0.0</upper_left_color>
         <upper_left_color>1.0</upper_left_color>
      </config_app>
   </elements>
</configuration>
```

The `configCanHandle()` Member Function

The implementation of `configCanHandle()` is located in `ConfigApp.cpp`. It will be invoked whenever a new configuration element is found, and its job is to tell the JCCL Configuration Manager if it can be handled or not. The full implementation is as follows:

```
bool ConfigApp::configCanHandle(jccl::ConfigElementPtr element)
{
   const std::string my_type("config_app");

   return (my_type == element->getID());
}
```

We define the element type that the application knows how to handle (named “ConfigApp”, as shown above), and we return `true` if that type matches the type of the newly loaded configuration element.

The `configAdd()` Member Function

When a configuration element matching our type is loaded, the method `configAdd()` is invoked. The parameter passed in is the full configuration element that was loaded. The job of `configAdd()` is to read the information from the configuration element and configure the application based on what is read. In this specific case, we will read color information used for coloring the ground polygon in the scene. A partial implementation of `ConfigApp::configAdd()` follows:

```
bool ConfigApp::configAdd(jccl::ConfigElementPtr element)
{
   const std::string ll_color("lower_left_color");

   // Get the color settings for the lower left corner (ll).
```

```
   for ( int i = 0; i < element->getNum(ll_color); ++i )
   {
      mSurfaceColor_ll[i] = element->getProperty<float>(ll_color, i);
   }

   // Get the color settings for the other three corners ...
}
```

Note that we are reading the values from the property named “lower_left_color.” While we know that there are three (3) values to read, we ask the configuration how many values it has just to be safe.

The `draw()` Member Function

For this application, the draw() member function is not terribly interesting. It renders the ground, and the colors for the four vertices are set using the member variables. Of course, the values for these member variables are set using the configuration file, and that is the interesting aspect.

Exercise

Chapter 12. Extending VR Juggler

VR Juggler can be extended in ways beyond the application object. In this chapter, we review points of extensibility briefly and provide pointers to more complete information for interested readers.

Device Drivers

Please refer to the Gadgeteer [http://www.vrjuggler.org/gadgeteer/docs.php] *Device Driver Authoring Guide* for a comprehensive description of how to write device drivers for use with VR Juggler applications.

Custom Simulators

To learn about how to add custom simulators to VR Juggler, please refer to the VR Juggler *Extension Guide*.

Simulator Components

The basic simulator for the OpenGL Draw Manager uses draw functors to render the simulator components including the head and the wand. Users can provide their own draw functors to customize the look of the basic simulator without going to the trouble of writing a custom simulator.

Chapter 13. Advanced Topics

In this chapter, we address topics that are considered to be for advanced uses of VR Juggler. This is not to say that they are difficult concepts or hard-to-use features. Rather, they are not things that the average VR Juggler application programmer will do or even need to be aware of when writing VR software. These are important, interesting, and worthwhile topics nonetheless.

Customizing Render Thread Processor Affinity

VR Juggler 2.2.1 introduced the feature of adjustable render thread processor affinity for Linux and IRIX users of the OpenGL Draw Manager. The default mechanism for defining processor affinity is based on an environment variable and the use of a simple round-robin scheme for the case of having more graphics pipes than processors. More tailored uses of render thread processor affinity can be achieved by giving the OpenGL Draw Manager a custom affinity assignment algorithm. That will be the topic of this section.

The OpenGL Draw Manager class `vrj::GlDrawManager` has a method named `setCpuAffinityStrategy()` that takes as its argument a callable object. This is encapsulated using a Boost.Function object of type boost::function<int (const unsigned int)>, which in English is a function that takes a single constant unsigned integer (the pipe identifier) and returns a signed integer (the processor identifier). Anything that can be encapsulated by a Boost.Function object can be used. That includes a pointer to a C function, a pointer to a static C++ class member function, a value returned by `boost::bind()`, or an instance of a class that overloads `operator()`. This usage implements the Strategy Pattern where functionality is plugged in at run time. The default behavior is to use an instance of `vrj::CpuAffinityFromEnv`, assigned in the `vrj::GlDrawManager` constructor.

The job of this callable object is to map the pipe identifier to the processor to which affinity will be assigned for the render thread of the pipe. The processor identifier is a signed integer so that a negative value can be used to indicate that no affinity should be assigned for the render thread of the pipe. The exact means by which it accomplishes this can vary, and that is the power of the Strategy Pattern in this context.

To provide a customized processor affinity assignment algorithm, `vrj::GlDrawManager::setCpuAffinityStrategy()` must be invoked at the correct time. The OpenGL Draw Manager uses whatever affinity assignment algorithm it has as soon as it determines that `vrj::GlPipe` objects need to be created. This is done as part of the VR Juggler run-time reconfiguration process. Therefore, to guarantee that all pipes use the customized algorithm, `vrj::GlDrawManager::setCpuAffinityStrategy()` must be called before calling `vrj::Kernel::start()`, the method that starts the VR Juggler kernel control loop. Since the kernel control loop is normally started in the application `main()` function, the most likely place for a call to `vrj::GlDrawManager::setCpuAffinityStrategy()` is also in `main()`. Of course, this can vary depending on how VR Juggler is being used, so the most important thing to understand is that it must happen *before* `vrj::Kernel::start()` is called.

Making a Custom `NSApplication` Delegate on Mac OS X

The Cocoa support for VR Juggler is designed to take advantage of the flexibility of Cocoa application design and, more generally, the flexibility of the Objective-C language. Specifically, VR Juggler application programmers targeting Mac OS X have a unique opportunity to customize the behavior of VR Juggler application behavior. Behind the scenes, `vrj::CocoaWrapper` registers a delegate, an instance of the Objective-C class `VRJBasicDelegate`, with the `NSApplication` singleton. Messages from `NSApplication` are received and handled by this object. VR Juggler application programmers can

tailor the handling of these messages by creating and using a custom `NSApplication` delegate.

The full details of delegates and customizing the behavior of `NSApplication` are well beyond the scope of this document. Readers are referred to Apple's documentation for Cocoa [http://developer.apple.com/referencelibrary/GettingStarted/GS_Cocoa/index.html] and `NSApplication` class reference [http://developer.apple.com/documentation/Cocoa/Reference/ApplicationKit/Classes/NSApplication_Class/Reference/Reference.html]. We will proceed from here under the assumption that readers are already familiar with the relevant and necessary topics including, but not limited to, Objective-C, Objective-C++, and `NSApplication`.

In a typical Cocoa application, the delegate would be given to the `NSApplication` singleton instance by sending that object the `-setDelegate:` message. Doing so, however, will interfere with the proper functionality of VR Juggler. Thus, the registration mechanism is done using the `VRJDelegateClass` property in `Contents/Resources/vrjuggler.plist`. The default value of this property is `VRJBasicDelegate`. By changing this to the name of the custom class, `vrj::CocoaWrapper` will instantiate, register, and use the custom class instead. (Being able to do this is a simple and elegant feature of Objective-C.)

There are two ways to create a custom delegate class. The class can either stand on its own as a subclass of `NSObject`, or it can derive from `VRJBasicDelegate`. The decision about which approach to use depends on whether complete or partial customization is desired. For this discussion, we will focus on complete customization—that is, creating a new delegate type by deriving from `NSObject`.

> It is worth pointing out that `VRJBasicDelegate` is not necessarily designed to be extended. It can be customized, but mainly, this customization is done by overriding methods to *replace* the behavior of the base class rather than to extend it. Refer to the implementation of `VRJBasicDelegate` to get a better idea of the implications of overriding it for the purposes of customization. The code can be found in `modules/vrjuggler/vrj/Kernel/VRJBasicDelegate.mm` and `modules/vrjuggler/vrj/Kernel/VRJBasicDelegate.h`.

As noted, the job of the delegate is to respond to messages from `NSApplication`. The messages that are delivered to the delegate are documented in the `NSApplication` reference, and we will not duplicate that information here. Instead, we will concentrate on a small subset of these messages related to document loading. The purpose is twofold. First, this explanation helps in understanding the implementation of `VRJBasicDelegate`, thereby clarifying options for customizing the behavior of that class. Second, document loading is a prominent topic in Mac OS X application development, and it is likely to be of the most interest to VR application programmers, too.

To get things under way, have a glance at Example 13.1, "`MyDelegate.mm`: Basic Delegate Implementation". It is a relatively simple class that uses Objective-C++ capabilities to create a simple bridge between `NSApplication` and `vrj::Kernel`. The documents being loaded in this case are VR Juggler configuration files, but it is not much of a leap to imagine the use of any other document type (i.e., data file used as input to the application). We will review each of the methods of this class.

Example 13.1. `MyDelegate.mm`: Basic Delegate Implementation

```
1 #import <Foundation/NSArray.h>
  #import <Foundation/NSString.h>
  #import <AppKit/NSApplication.h>

5 #include <vrj/Kernel/Kernel.h>
```

```
@interface MyDelegate : NSObject
{
   BOOL mLoadConfigs;
}
@end

@implementation MyDelegate
   -(id) init
   {
      mLoadConfigs = YES;
      return [super init];
   }

   -(void) setLoadConfigs:(BOOL) load
   {
      mLoadConfigs = load;
   }

   -(BOOL) applicationShouldTerminateAfterLastWindowClosed:(NSApplication*) se
   {
      // We return NO here because we have a different way of shutting down
      // the application. When vrj::Kernel::stop() is invoked, it will cause
      // the application run loop to stop by invoking
      // vrj::CocoaWrapper::stop().
      return NO;
   }

   -(void) applicationDidFinishLaunching:(NSNotification*) aNotification
   {
      // We're ready to allow windows to open!
      NSConditionLock* lock = gadget::InputAreaCocoa::getWindowLock();
      [lock unlock];
   }

   -(BOOL) application:(NSApplication*) theApplication
             openFile:(NSString*) file
   {
      if ( mLoadConfigs )
      {
         vrj::Kernel::instance()->loadConfigFile([file UTF8String]);
      }

      return YES;
   }

   -(void) application:(NSApplication*) theApplication
              openFiles:(NSArray*) files
   {
      if ( mLoadConfigs )
      {
         const int count = [files count];
         for ( int i = 0; i < count; ++i )
         {
            NSString* file = [files objectAtIndex:i];
            vrj::Kernel::instance()->loadConfigFile([file UTF8String]);
         }
      }
   }
@end
```

Data Members

The class `MyDelegate` has a single data member, `mLoadConfigs`, that determines how instances of this class will respond to certain messages sent by `NSApplication`. The value will be changed if `vrj::CocoaWrapper` sends the `-setLoadConfigs:` message as a result of the property `VRJConfigHandling` being set in `Contents/Resources/vrjuggler.plist`.

Designated Initializer

Every `NSObject` subclass has a designated initializer. In the case of the VR Juggler `NSApplication` delegate, this is the basic `-init` method. The current usage of the application delegate requires that this method be the designated initializer, meaning that it is the only initializer that will be invoked upon creating the delegate instance. This limitation may be fixed in a future version of `vrj::CocoaWrapper`. If the custom delegate does not accept the -init message, then the inherited `NSObject` version will be used.

In the meantime, this is where all object initialization takes place. What this means exactly is up to the delegate author. In this case, we set the data member `mLoadConfigs` to `YES`, send our base class the `-init` message, and return the result. This last step (a widely used convention) is what must be done by any custom delegate implementation of `-init`.

`-setLoadConfigs:`

This method is a unique aspect of the delegate as it is handled by `vrj::CocoaWrapper`. Specifically, `vrj::CocoaWrapper` will examine `Contents/Resources/vrjuggler.plist` to see if it contains the property `VRJConfigHandling`. If it does, it will send this message to the delegate instance with the value as it is set in the property list. This happens before the delegate object is handed off to the `NSApplication` singleton. The consequences of setting and using the `VRJConfigHandling` property were described in the section called “Application Execution”.

A custom delegate does not have to implement this method. If it does not, the Objective-C runtime will print a warning on the console stating that the object does not receive this message. (A future version of `vrj::CocoaWrapper` may examine the delegate interface to determine if it receives the message just to avoid having that message printed.) It is up to the delegate author to determine if implementing this method is necessary and, if so, how it should behave. In our example, we set `mLoadConfigs` to the value of the parameter, just as `VRJBasicDelegate` does.

`-applicationShouldTerminateAfterLastWindowClosed:`

This method is used by `NSApplication` to allow customization of application shutdown. In the case of an application delegate instantiated and used by `vrj::CocoaWrapper`, it is critical that this method return the value `NO`. The VR Juggler kernel knows about `vrj::CocoaWrapper` and knows how to shut it down when the time is right. Because of that, we want the kernel to control the termination process.

`-applicationDidFinishLaunching:`

When the application finishes launching, NSApplication sends this message to its delegate. In the case of VR Juggler, this is when we know that it is safe to allow threads for Gadgeteer input windows and VR Juggler graphics pipes to open windows. Until now, any such threads have been blocking on the global window lock, acquired when `vrj::CocoaWrapper` was instantiated. When the delegate receives this message, it is time to release that lock. Other operations can be performed in this method implementation, but this step is absolutely critical.

-application:openFile:

Now, we get to one of the methods used by NSApplication to tell its delegate when to open a single data file. Remember that our simple delegate just handles VR Juggler configuration files. Moreover, it is assuming (probably unsafely) that the only files that it will be given to load are VR Juggler configuration files.

In any case, NSApplication sends this message to its delegate at well-defined times. The response of our simple delegate is based on the value of mLoadConfigs. If it is true (set to YES), that means that our delegate will load configuration files as a result of the user double-clicking on .jconf files in the Finder or through the use of the **open** command. Loading of configuration files is done through the usual means of passing the file name to the vrj::Kernel instance. Note that we convert the NSString Unicode value to a UTF8 string because that is what vrj::Kernel::loadConfigFile() knows how to handle. A future version of VR Juggler may have proper Unicode support so that this step is not necessary.

Note that this method returns YES regardless of the value of mLoadConfigs. This has the effect of making NSApplication think that the file was loaded even if it was ignored. In some cases, it may make more sense to return NO instead. As with just about everything in the custom delegate, the exact response to this message is up to the author.

-application:openFiles:

This method is the multi-file counterpart to -application:openFile:. The behavior is essentially the same except that it iterates over the contents of the given NSArray object and calls vrj::Kernel::loadConfigFile() for each of the contained NSString objects.

Defining Custom Cocoa/VR Juggler Bridging on Mac OS X

More intrepid programmers may find that writing a custom NSApplication delegate is not sufficient for customizing VR Juggler application behavior on Mac OS X. In that event, the customization can be even more extensive by replacing the use of vrj::CocoaWrapper altogether. For all intents and purposes, vrj::CocoaWrapper is grafted on to vrj::Kernel to provide the mediator between the competing desires of NSApplication and vrj::Kernel to control application execution[1]. vrj::Kernel can be told not to use vrj::CocoaWrapper, thus providing an extension point where a different Cocoa/VR Juggler bridging can be used. This is done by calling the static method vrj::Kernel::setUseCocoaWrapper() with the parameter false *before* the first call to vrj::Kernel::instance().

In general, the programmer of this bridging code is free to do whatever makes sense, but there are a few caveats. First, every thread that is created by any Juggler component needs to have an NSAutoReleasePool created. This is done by registering thread start and thread exit methods with vpr::Thread.

Next, AppKit and Cocoa need to know that they are being used in a multi-threaded environment. They are made aware of this fact when the first NSThread object is created. VPR uses low-level POSIX threads on Mac OS X rather than using NSThread. Thus, a dummy thread needs to be created using NSThread. All it has to do is run an empty method and exit. The thread start callback is registered using vpr::Thread::addThreadStartCallback(), and the thread exit callback is registered using vpr::Thread::addThreadExitCallback(). The thread start callback must allocate and initialize a thread-specific instance of NSAutoReleasePool, and the thread exit callback must re-

[1]This is not a strictly accurate description of the design because vrj::Kernel has to know about vrj::CocoaWrapper and call its methods at certain times. A true mediator would sit above vrj::Kernel and vrj::CocoaWrapper and coordinate their interaction to prevent such tight coupling, but introducing that sort of design would complicate things unnecessarily.

lease its reference to the `NSAutoReleasePool` instance created by the thread start callback. These callbacks have to be registered *before* any `vpr::Thread` objects are created.

Finally, the global window lock must be held until it is safe to open input and graphics windows. This is necessary because Gadgeteer and VR Juggler open windows in threads other than the primordial, and AppKit does not handle that well. This lock (an instance of `NSConditionLock`) is retrieved using the static method `gadget::InputAreaCocoa::getWindowLock()`. The job of the bridging code is to acquire the lock as soon as it can. Specifically, it should do it *before* creating the `NSApplication` singleton object by sending the `NSApplication` class the `+sharedApplication` message. The global lock must be released once we know that the application has finished launching. If a delegate is used with the `NSApplication` singleton, the lock should be released when the delegate receives the `-applicationDidFinishLaunching:` message.

Beyond that, the programmer is free to do as s/he wishes in the implementation of the Cocoa/VR Juggler bridging code. Referring to the implementation of `vrj::CocoaWrapper` is highly recommended (see `modules/vrjuggler/vrj/Kernel/CocoaWrapper.mm` and `modules/vrjuggler/vrj/Kernel/CocoaWrapper.h` in the source tree), but its exact behavior does not have to be replicated. The points presented above are the most important things that must be done by any implementation of bridging code.

Ultimately, all of this work is done to help VR Juggler get along with Cocoa. VR Juggler makes heavy use of multi-threading whereas AppKit and Cocoa tend to work best in single-threaded environments—or at in least environments where the primordial thread is receiving and dispatching all window system events. Things are just not that simple with VR Juggler, and thus the bridge between these frameworks plays a vital role.

Part IV. Appendices

Table of Contents

Appendix A. GNU Free Documentation License

Version 1.2, November 2002

FSF Copyright note

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sec-

tions then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you

as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

GNU FDL Modification Conditions

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the

title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the

combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> **Sample Invariant Sections list**
>
> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> **Sample Invariant Sections list**
>
> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Glossary of Terms

A

application object	An instance of a C++ class that is given to the VR Juggler kernel for execution. Such objects contain the visuals and the interactions that make up a virtual world.

C

callback	In the C programming language, a function pointer that is invoked to "call back" to some other code. Usually, a function pointer is passed as an argument to some function that stores the pointer for later use.
context-specific data	In OpenGL terms, some information that is associated with a specific OpenGL context.
critical section	In multi-threaded programming, a block of code that reads from or writes to data that is shared across multiple threads.

D

daemon	In UNIX-based operating systems, a background process that performs some job continuously while the computer is running. For example, a web server runs a daemon (often called **httpd**) that manages incoming HTTP requests.
device interface	A high-level feature provided by Gadgeteer for getting easy access to a type-specific device proxy. For each input device type, there is one device proxy type, and for each device proxy type, there is one device interface type. For example, digital input devices are of type `gadget::Digital`. Digital input data is accessed by VR Juggler application objects through a `gadget::DigitalProxy`. The proxy reference is most easily acquired through a `gadget::DigitalInterface` object.
display context	The location to which OpenGL rendering commands draw.

F

frame	In computer graphics terms, one iteration of a rendering loop. In VR Juggler, a frame is one complete pass through an application object's methods, beginning with `vrj::App::preFrame()` and ending with `vrj::App::postFrame()`. Methods called in between include `vrj::App::intraFrame()` and methods that are specific to a given graphics API.

G

GUID — Globally unique identifier that is generated using various properties of a machine and a random number generator. It has been proven that it is statistically impossible for the same number to be generated twice within the average human lifetime.

I

interface — An interface is a collection of operations used to specify a service of a class or a component.

O

OpenGL rendering context — An OpenGL state machine associated with a display window.

P

proxy — An intermediary that passes information between two parties.

S

smart pointer — In C++ terminology, a smart pointer is a pointer-like object where the pointer dereference operators (-> and *) are overloaded to perform special functionality. This takes advantage of C++ operator overloading to hide extra processing steps behind a familiar syntax.

stupefied proxy — A device proxy that has no device. Such a proxy always returns the same data because it cannot query new data from an input device. Proxies become stupefied when there is an error in the configuration or when the proxied device is unavailable for some reason.

T

thread of control — In a multi-threaded application, a single sequence of execution. Each thread in a multi-threaded application has its own thread of control that can execute in parallel with the other threads of control.

triple buffering — An extension to double buffering that uses three data buffers instead of two. Triple buffering minimizes the time that two threads have to wait to access shared data. At most, one thread will wait while another thread copies no more than four bytes of data. That is, regardless of the size of a single data buffer, the amount of memory copied to swap buffers is no more than four bytes (the size of a 32-bit memory address).

V

virtual platform — An abstraction of the operating system and hardware (both the com-

puter architecture and the input devices) that comprise a VR system. A virtual platform provides a cross-platform and cross-VR system layer upon which portable VR applications can be written.

Index

E

F

G

I

J

M

O

P

R

S

T

V