

VR Juggler

The Programmer's Guide

VR Juggler: The Programmer's Guide

Published \$Date: 2002/09/24 17:31:36 \$

Table of Contents

I. Introduction	1
1. Getting Started	4
Necessary Experience	4
Required Background	4
Other VR Software Tools	4
Organization	4
2. Application Basics	6
Application Object Overview	6
No main()—"Don't call me, I'll call you"	6
Application Objects Derive from Base Classes for Specific Graphics APIs	6
Writing an Application Means Filling in the Blanks	6
Benefits of Application Objects	6
Allow for Run-Time Changes	7
Low Coupling	7
Allows Implementation Changes	7
VR Juggler Startup	7
No main()—Sort Of	7
Structure of a main() Function	7
Kernel Loop	8
Definition of a Frame	8
Base Application Object Interface	9
Initialization	9
Frame Functions	10
Draw Manager-Specific Application Classes	11
OpenGL Application Class	11
OpenGL Performer Application Class	11
3. Helper Classes	12
The vjVec3 and vjVec4 Helper Classes	12
High-Level Description	12
Using vjVec3 and vjVec4	13
Creating Vectors and Setting Their Values	13
Inversion (Finding the Negative of a Vector)	13
Normalization	14
Length Calculation	14
Multiplication by a Scalar	14
Division by a Scalar	15
Converting to an OpenGL Performer Vector	15
Assignment	16
Equality/Inequality Comparison	16
Dot Product	16
Cross Product (vjVec3 only)	17
Addition	17
Subtraction	17
Full Transformation by a Matrix	17
Partial Transformation by a Matrix	18
The Gory Details	18
The vjMatrix Helper Class	18
High-Level Description	19
Using vjMatrix	20
Creating Matrices and Setting Their Values	20
Assignment	21
Equality/Inequality Comparison	21
Transposing	22

Finding the Inverse	22
Addition	22
Subtraction	23
Multiplication	23
Scaling by a Scalar Value	24
Making an Identity Matrix Quickly	25
Zeroing a Matrix in a Single Step	25
Making an XYZ, a ZYX, or a ZXY Euler Rotation Matrix	25
Constraining Rotation About a Specific Axis or Axes	26
Making a Matrix Using Direction Cosines	26
Making a Matrix from a Quaternion	26
Making a Rotation Transformation Matrix About a Single Axis	27
Making a Translation Transformation Matrix	28
Making a Scale Transformation Matrix	29
Extracting Specific Transformation Information	30
Converting to an OpenGL Performer Matrix	30
The Gory Details	31
The <code>vjDeviceInterface</code> Helper Class	32
High-Level Description	32
Using <code>vjDeviceInterface</code>	32
The Gory Details	33
The <code>vjProxy</code> Helper Class	34
High-Level Description	34
Using <code>vjProxy</code>	34
The Gory Details	34
II. Application Programming	35
4. Writing Applications	38
Application Review	38
Basic Application Information	38
Draw Manager-Specific Application Classes	38
Getting Input	38
How to Get Input	39
Where to Get Input	39
Tutorial: Getting Input	40
OpenGL Applications	41
OpenGL Drawing: <code>vjGApp::draw()</code>	42
Tutorial: Drawing a Cube with OpenGL	42
Context-Specific Data	43
Using Context-Specific Data	45
Context-Specific Data Details	47
Tutorial: Drawing a Cube using OpenGL Display Lists	47
OpenGL Performer Applications	49
Scene Graph Initialization: <code>vjPfApp::initScene()</code>	49
Scene Graph Access: <code>vjPfApp::getScene()</code>	49
Tutorial: Loading a Model with OpenGL Performer	49
Other <code>vjPfApp</code> Methods	51
VTK Applications	53
5. Porting to VR Juggler from the CAVELib	54
The Initialize, Draw, and Frame Routines	54
In CAVELib	54
In VR Juggler	54
Getting Input from Devices	55
In CAVELib	55
In VR Juggler	55
Configuration	56
In CAVELib	56
In VR Juggler	56
Important Notes	56

Shared Memory	56
OpenGL Context-Specific Data	56
Source Code	56
The Form of a Basic CAVELib Program	57
The Form of a Basic VR Juggler Program	57
6. Porting to VR Juggler from GLUT	59
Window Creation and Management	59
The Initialize, Draw, and Frame Routines	59
In GLUT	59
In VR Juggler	59
Getting Input from Devices	60
In GLUT	60
In VR Juggler	60
Configuration	61
In GLUT	61
In VR Juggler	61
Important Notes	61
Shared Memory	61
OpenGL Context-Specific Data	61
Source Code	62
The Form of a Basic GLUT Program	62
The Form of a Basic VR Juggler Program	62
III. Advanced Topics	64
7. System Interaction	66
8. Multi-threading	67
Techniques	67
Helper Classes	67
Using the vjThread Interface	67
Using the vjBaseThreadFunctor Interface	72
Using the vjSemaphore Interface	75
Using the vjMutex Interface	76
9. Run-Time Reconfiguration	79
How Run-Time Reconfiguration Works	79
Reasons to Use Run-Time Reconfiguration	79
Using Run-Time Reconfiguration in an Application	79
Application Tutorial	81
10. Adding Device Drivers to VR Juggler	82
In-Depth Driver Guide	82
Implementing the Device Driver	82
Register the Device Driver with VR Juggler	85
Device Driver Configuration	85
Configuration Files	85
Writing Code that Accepts the Configuration	86
Example Code	86
Index	89

List of Figures

2.1. vjApp hierarchy	
2.2. Kernel loop sequence	
2.3. Application object interface	
2.4. vjGLApp interface extensions to vjApp	
2.5. vjPfApp interface extensions to vjApp	
4.1. VR Juggler kernel control loop	
4.2. vjGLApp application class	
4.3. VR Juggler OpenGL system	
4.4. vjPfApp application class	

List of Tables

3.1. Row-major access indices	31
3.2. Column-major access indices	31
4.1. Tutorial Overview	40
4.2. Tutorial Overview	42
4.3. Tutorial Overview	47
4.4. Tutorial Overview	49

List of Examples

4.1. Initializing context-specific data	46
---	----

Part I. Introduction

Table of Contents

1. Getting Started	4
Necessary Experience	4
Required Background	4
Other VR Software Tools	4
Organization	4
2. Application Basics	6
Application Object Overview	6
No main()—"Don't call me, I'll call you"	6
Application Objects Derive from Base Classes for Specific Graphics APIs	6
Writing an Application Means Filling in the Blanks	6
Benefits of Application Objects	6
Allow for Run-Time Changes	7
Low Coupling	7
Allows Implementation Changes	7
VR Juggler Startup	7
No main()—Sort Of	7
Structure of a main() Function	7
Kernel Loop	8
Definition of a Frame	8
Base Application Object Interface	9
Initialization	9
Frame Functions	10
Draw Manager-Specific Application Classes	11
OpenGL Application Class	11
OpenGL Performer Application Class	11
3. Helper Classes	12
The vjVec3 and vjVec4 Helper Classes	12
High-Level Description	12
Using vjVec3 and vjVec4	13
Creating Vectors and Setting Their Values	13
Inversion (Finding the Negative of a Vector)	13
Normalization	14
Length Calculation	14
Multiplication by a Scalar	14
Division by a Scalar	15
Converting to an OpenGL Performer Vector	15
Assignment	16
Equality/Inequality Comparison	16
Dot Product	16
Cross Product (vjVec3 only)	17
Addition	17
Subtraction	17
Full Transformation by a Matrix	17
Partial Transformation by a Matrix	18
The Gory Details	18
The vjMatrix Helper Class	18
High-Level Description	19
Using vjMatrix	20
Creating Matrices and Setting Their Values	20
Assignment	21
Equality/Inequality Comparison	21
Transposing	22
Finding the Inverse	22

Addition	22
Subtraction	23
Multiplication	23
Scaling by a Scalar Value	24
Making an Identity Matrix Quickly	25
Zeroing a Matrix in a Single Step	25
Making an XYZ, a ZYX, or a ZXY Euler Rotation Matrix	25
Constraining Rotation About a Specific Axis or Axes	26
Making a Matrix Using Direction Cosines	26
Making a Matrix from a Quaternion	26
Making a Rotation Transformation Matrix About a Single Axis	27
Making a Translation Transformation Matrix	28
Making a Scale Transformation Matrix	29
Extracting Specific Transformation Information	30
Converting to an OpenGL Performer Matrix	30
The Gory Details	31
The <code>vjDeviceInterface</code> Helper Class	32
High-Level Description	32
Using <code>vjDeviceInterface</code>	32
The Gory Details	33
The <code>vjProxy</code> Helper Class	34
High-Level Description	34
Using <code>vjProxy</code>	34
The Gory Details	34

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. To those in this category, fear not—VR Juggler is very simple to use after getting through the initial learning stages. It is a very 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 of transformations that must be applied to three-dimensional 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 five main chapters:

1. Introduction
2. Application basics: The introduction to the key VR Juggler application development concept, application ob#

jects.

3. Common helper classes: A description of useful classes provided to simplify writing applications.
4. 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).
5. 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 interfaces¹ 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 (`vjApp`) and a base class for each Draw Manager interface (`vjPfxApp`, `vjGlApp`, etc.). See Figure 2.1 for a visual representation of the application interface hierarchy. The kernel interface defined in `vjApp` specifies methods for initialization, shutdown, and execution of the application. The Draw Manager interfaces specified in the `vj*App` classes define the API-specific functions necessary to render the virtual environment. For example, a Draw Manager interface could have functions for drawing the scene and for initializing context-specific information.

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.

Benefits of Application Objects

As stated earlier, the most common approach for VR application development is one where the application defines

¹An interface is a collection of operations used to specify a service of a class or a component.

the `main()` function. That `main()` function in turn calls library functions when needed. 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 possible to load and unload applications dynamically. When the virtual platform initializes, it waits for an application to be passed to it. When the application is given to the VR Juggler kernel at run time, the kernel performs a few initialization steps and then executes the application.

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.

VR Juggler Startup

No `main()`—Sort Of

Previously, we explained how VR Juggler applications do not have a `main()` function. 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 VR Juggler 1.0 applications, there is a `main()`, but it only starts the VR Juggler kernel and gives the kernel the application to run.

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 <simpleApp.h>
   int main (int argc, char* argv[])
   {
5    vjKernel* kernel = vjKernel::instance(); // Get the kernel
❶    simpleApp* app   = new simpleApp();      // Create the app object
❷
```

```

kernel->loadConfigFile(...);           // Configure the kernel
❸ kernel->start();                       // Start the kernel thread
❹
10 kernel->setApplication(app);         // Give application to kernel
❺

    while ( ! exit )
    {
15     // sleep
    }
}

```

- ❶ This line finds (and may create) the VR Juggler kernel. The kernel reference is stored in the handle so that we can use it later.
- ❷ We instantiate a copy of the user application object (`simpleApp`) here. Notice that we include the header file that defines the `simpleApp` class.
- ❸ 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.
- ❹ 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.
- ❺ 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.

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. 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.

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; it makes up all the lines within the “while(running)” 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 chapter, we provide a brief overview of the member functions from the base VR Juggler application interface. This interface is defined by `vjApp`, and the member functions are shown in Figure 2.3. Refer to Figure 2.2 for a visual presentation of the order in which the methods are invoked.

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

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

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

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.

`vjApp::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.

`vjApp::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 not normally used 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.

`vjApp::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 immediately 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).

`vjApp::intraFrame()`

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

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.

`vjApp::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 `vjApp::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 `vjApp` 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 `vjApp` class are shown in Figure 2.4. In the following, we describe the method `vjGLApp::draw()`, the most important element of the interface. More details about the `vjGLApp` class are provided in the section called “OpenGL Applications”, found in Chapter 4, *Writing Applications*.

`vjGLApp::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. The following is a description of only two methods in the `vjPfApp` interface. More detailed discussion on this class is provided in the section called “OpenGL Performer Applications”, found in Chapter 4, *Writing Applications*.

`vjPfApp::initScene()`

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

`vjPfApp::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 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. In particular, special attention is paid to the VR Juggler Input Manager device interfaces and proxies.

The `vjVec3` and `vjVec4` Helper Classes

This section is intended to provide an introduction to how the helper classes `vjVec3` and `vjVec4` work and how they 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 classes `vjVec3` and `vjVec4` are designed to work the same way as three- and four-dimensional mathematical vectors. That is, a `vjVec3` object can be thought of as a vector of the form $\langle x, y, z \rangle$. Similarly, a `vjVec4` can be thought of as a vector of the form $\langle x, y, z, w \rangle$. 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 used typically 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. VR Juggler's vectors retain this generality and can be used wherever vectors come in handy.

`vjVec3` and `vjVec4`, 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 (`vjVec3` only)
- Addition
- Subtraction

Knowing this and keeping in mind that `vjVec3` and `vjVec4` can be thought of at this high level, using them should be a snap.

Using `vjVec3` and `vjVec4`

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 either `Math/vjVec3.h`, `Math/vjVec4.h`, or both as necessary. 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 `vjVec3`s, but the example is equally applicable to `vjVec4`. To create a `vjVec3`, use the default constructor which initializes the vector to `<0.0, 0.0, 0.0>`:

```
vjVec3 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:

```
vjVec3 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 VR Juggler 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:

```
vjVec3 vec1(1.0, 1.5, -1.0), vec2;  
vec2 = -vec1;
```

The vector `vec2` now has the value $\langle -1.0, -1.5, 1.0 \rangle$. 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:

```
vjVec3 vec1(1.0, 1.5, -1.0);  
vec1.normalize();
```

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 ( vec.isNormalized() )  
{  
    // Go here if vec is normalized  
}
```

This only works with `vjVec3s`, however.

Length Calculation

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

```
vjVec3 vec1(1.0, 1.5, -1.0);  
float length;  
  
length = vec1.length();
```

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, the `vjVec3` class (but not `vjVec4`) provides a method called `lengthSquared()` that returns the length of the vector without calculating the square root.

Multiplication by a Scalar

The VR Juggler 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:

```
vjVec3 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:

```
vjVec3 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:

```
vjVec3 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:

```
vjVec3 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

SGL's OpenGL Performer likes to work with its own `pfVec3` class, and to facilitate the use of it with `vjVec3`, two conversion functions are provided for converting a `vjVec3` to a `pfVec3` and vice versa. The first works as follows:

```
vjVec3 vj_vec;
pfVec3 pf_vec;

// Do stuff to vj_vec...

pf_vec = vjGetPfVec(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 `vjVec3`, do the following:

```
pfVec3 pf_vec;
vjVec3 vj_vec;

// Do stuff to pf_vec...

vj_vec = vjGetVjVec(pf_vec);
```

Here again, `pf_vec` is passed by reference for efficiency, and `vj_vec` gets a copy of a `vjVec3`. Both of these

functions are found in the header `Kernel/Pf/vjPfUtil.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:

```
vjVec3 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 types of assignments of VR Juggler 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):

```
vjVec3 vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
if ( vec1.equal(vec2) )
{
    // Go here if vec1 and vec2 are equal.
}
```

or

```
vjVec3 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

```
vjVec3 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. VR Juggler's 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:

```
vjVec3 vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0);
float dot_product;

dot_product = vec1.dot(vec2);
```

Now, `dot_product` has the value 4.0.

Cross Product (vJVec3 only)

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

```
vjVec3 vec1(1.0, 1.5, -1.0), vec2(1.5, 1.0, -1.0), vec3;  
vec3 = vec1.cross(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.

```
vjVec3 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>`.

```
vjVec3 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.

```
vjVec3 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>`.

```
vjVec3 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 method `xformFull()` does this job. For the following example, assume that there is a `vjMatrix` transformation matrix `xform_mat`:

```
vjVec3 vec(1.0, 1.0, 1.0);
```

```
vec.xformFull(xform_mat);
```

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

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

Partial Transformation by a Matrix

Besides full transformations, it is possible to apply all transformations in a matrix except translation. (The usefulness of this is left as an exercise for the reader.) Again assume that there is a `vjMatrix` transformation matrix `xform_mat`:

```
vjVec3 vec(1.0, 1.0, 1.0);  
vec.xformVec(xform_mat);
```

In this case, all transformations in `xform_mat` except translation will be applied to `vec`. The naming convention here is not terribly helpful, but this documentation is intended to get around that.

The Gory Details

The details behind `vjVec3` and `vjVec4` 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 public member variable `vec`. For example, this access can be used in the following way:

```
vjVec3 pos(4.0, 1.0982, 10.1241);  
glVertex3fv(pos.vec);
```

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 `vec` member variable 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 `vec` follows:

```
vjVec4 my_vec;  
  
my_vec.vec[0] = 4.0;  
my_vec.vec[1] = 1.0982;  
my_vec.vec[2] = 10.1241;  
my_vec.vec[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 `vjMatrix` Helper Class

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

High-Level Description

Abstractly, `vjMatrix` 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 `vjMatrix` 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 `vjMatrix`. The class provides a degree of convenience not found with a normal C++ array, especially when programming with OpenGL. The complications surrounding the `vjMatrix` 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 `vjMatrix`.

As a representation of mathematical matrices, `vjMatrix` 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 `vjMatrix` class at the C++ level.

Using `vjMatrix`

Keeping the idea of a normal mathematical matrix in mind, we are now ready to look at the C++ use of the `vjMatrix` 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 `vjMatrix`, make sure to include `Math/vjMatrix.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 `vjMatrix`, the default constructor can be used. It initializes the matrix to be an identity matrix:

```
vjMatrix 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 `vjMatrix` 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:

```
vjMatrix 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 `vjMatrix` object, programmers can use either the overloaded `[]` operator or the `over#`

loaded () operator. The overloaded [] operator returns the specified row of the `vjMatrix`, 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:

```
vjMatrix 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:

```
vjMatrix 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 `vjMatrix` to another happens using the normal = operator as follows:

```
vjMatrix 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);
vjMatrix 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):

```
vjMatrix 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);
vjMatrix 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.equal(mat2) )
{
    // Go here if mat1 and mat2 are equal.
}
```

or

```
vjMatrix 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);
vjMatrix 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

```

vjMatrix 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);
vjMatrix 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 . The code is then:

```

vjMatrix mat1;
vjMatrix 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.transpose(mat2);

```

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

Finding the Inverse

The transpose operation works conceptually as . The code is then:

```

vjMatrix mat1;
vjMatrix 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.invert(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 `vjMatrix`: 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 `vjMatrix` objects.

```

vjMatrix 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);
vjMatrix 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);
vjMatrix mat3;

```

Using the `add()` method:

```

mat3.add(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 `vjMatrix`: 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 `vjMatrix` objects.

```
vjMatrix 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);
vjMatrix 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);
vjMatrix mat3;
```

Using the `sub()` method:

```
mat3.sub(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 `vjMatrix`: using the `mult()` method or using the overloaded * operator. We recommend the use of the `mult()` method, 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 `vjMatrix` objects.

```
vjMatrix 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);
vjMatrix 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);
vjMatrix mat3;
```

Using the `mult()` method:

```
mat3.mult(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:

```

vjMatrix 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);
vjMatrix 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.postMult(mat2);
  
```

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

```

vjMatrix 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);
vjMatrix 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:

```

vjMatrix 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);
vjMatrix 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.preMult(mat2);
  
```

Here, the conceptual operation is 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 `scale()` method or the overloaded `*` and `/` operators that take a single scalar value and returns a `vjMatrix`. 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 `scale()` method works as:

```

vjMatrix 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);
vjMatrix mat2;

mat2.scale(3.0, mat1);
  
```

Using the overloaded `*` operator works as:

```

vjMatrix 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);
vjMatrix mat2;

mat2 = mat1 * 3.0;
  
```

or

```

vjMatrix 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);
vjMatrix mat2;

mat2 = 3.0 * mat1;

```

Using the overloaded / operator works as:

```

vjMatrix 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);
vjMatrix mat2;

mat2 = mat1 / 3.0;

```

In all cases, the result of the scaling is stored (via a copy) in `mat2`.

Making an Identity Matrix Quickly

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

```

vjMatrix 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);

mat1.makeIdent();

```

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 `vjMatrix`, this can be done in one step:

```

vjMatrix 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);

mat1.zero();

```

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:

```

vjMatrix mat1;
float x_rot = 0.4, y_rot = 0.541, z_rot = 0.14221;

mat1.makeXYZEuler(x_rot, y_rot, z_rot);
mat1.makeZYXEuler(z_rot, y_rot, x_rot);
mat1.makeZXYEuler(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.

The rotation values can be read from a matrix that previously had one of the above operations applied to it. This is done as follows:

```
float x_rot, y_rot, z_rot;

mat1.makeXYZEuler(x_rot, y_rot, z_rot);
mat1.makeZYXEuler(z_rot, y_rot, x_rot);
mat1.makeZXYEuler(z_rot, x_rot, y_rot);
```

In real code, one would use only the call that is appropriate for extracting the rotation values from `mat`. The float variables are passed by reference to the method so that the rotation values can be returned in them.

Constraining Rotation About a Specific Axis or Axes

In a matrix that has rotations about more than one axis, it may be useful to get a transformation matrix that constrains the rotation to certain axes. A method is provided to do just this. It is called `constrainRotAxis()`, and it is used as follows (assume that `old_mat` is already defined as a transformation matrix with rotations about all three axes):

```
vjMatrix new_mat;

old_mat.constrainRotAxis(true, false, false, new_mat);
```

The result of this is that `new_mat` is a transformation matrix containing all of `old_mat`'s transformations with the exception that rotation is constrained about the x-axis alone. The first three arguments are Boolean values that specify the axis constraints for the x-, y-, and z-axes respectively. `new_mat` is passed by reference, and the results of the constrained transformations are stored in it.

Making a Matrix Using Direction Cosines

Creating a direction cosine matrix is another part of the `vjMatrix` interface. The method requires values for the x-, y-, and z-axes of the secondary coordinate system in terms of the first. These must be passed as objects of type `vjVec3`. Use of the method is shown below (assuming that `sec_x_axis`, `sec_y_axis`, and `sec_z_axis` are already defined and have appropriate values):

```
vjMatrix mat;

mat.makeDirCos(sec_x_axis, sec_y_axis, sec_z_axis);
```

The result is that `mat` becomes a direction cosine matrix.

Making a Matrix from a Quaternion

Converting a quaternion to its corresponding matrix is possible with `vjMatrix` objects. Two methods are provided to do this, each taking a different type of argument as the quaternion to be converted. The first takes a four-element array of floats:

```
vjMatrix mat;
float quat[4];

// Fill in quat...
```

```
mat.makeQuaternion(quat);
```

Here, `quat` is assigned its values appropriately. The other version takes a reference to a `vjQuat` object:

```
vjMatrix mat;
vjQuat quat;

// Fill in quat...

mat.makeQuaternion(quat);
```

The result in both cases is that `mat` is the transformation matrix represented by the quaternion.

Making a Rotation Transformation Matrix About a Single Axis

To make a rotation matrix where the rotation is about a single axis, a simple method is provided as part of the `vjMatrix` interface. It takes the rotation about the axis in degrees and the axis vector as a `vjVec3` object. Its use is as follows:

```
vjMatrix mat;
vjVec3 axis(1.0, 0.0, 0.0);

mat.makeRot(45.0, axis);
```

The `makeRot()` method causes `mat` to become a transformation matrix with only the given rotation. This rather boring example therefore illustrates making a transformation matrix with only a 45° rotation about the x-axis.

Further capabilities are available with the rotations. Similar to the `preMult()` and `postMult()` methods described earlier, one can perform pre-rotations and post-rotations in degrees about a given axis on a matrix. To do a pre-rotation, the code would look similar to the following:

```
vjMatrix mat1, mat2;
vjVec3 axis(1.0, 0.0, 0.0);

// Perform various transformations on mat2...

mat1.preRot(45.0, axis, mat2);
```

where `axis` is passed by reference. The result is that `mat1` is a transformation matrix assigned the value of multiplying the given rotation by `mat2` in that order. `mat2` is passed by reference to improve efficiency.

To do post-rotation, the code is similar:

```
vjMatrix mat1, mat2;
vjVec3 axis(1.0, 0.0, 0.0);

// Perform various transformations on mat2...

mat1.postRot(mat2, 45.0, axis);
```

where `axis` is passed by reference. The result in both cases is that `mat1` is a transformation matrix assigned the value of multiplying `mat2` by the given rotation in that order. `mat2` is passed by reference to improve efficiency. Note that for post-rotation, the matrix argument is given before the rotation arguments as a reminder that it comes first in the multiplication order.

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):

```
vjMatrix mat;  
vjVec3 trans(4.0, -4.231, 1.0);  
mat.makeTrans(trans);
```

or

```
vjMatrix mat;  
mat.makeTrans(4.0, -4.231, 1.0);
```

The only difference between these two is that the first takes a reference to a `vjVec3` object specifying the translation.

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

```
vjVec3 trans(4.0, -4.231, 1.0);  
mat.setTrans(trans);
```

or

```
mat.setTrans(4.0, -4.231, 1.0);
```

There are further extensions to translations. Similar to the `preMult()` and `postMult()` methods described earlier, one can perform pre-translations and post-translations on a matrix. To do a pre-translation, the code would look similar to the following:

```
vjMatrix mat1, mat2;  
vjVec3 trans(4.0, -4.231, 1.0);  
  
// Perform various transformations on mat2...  
mat1.preTrans(trans, mat2);
```

(where `trans` is passed by reference) or

```
vjMatrix mat1, mat2;  
  
// Perform various transformations on mat2...  
mat1.preTrans(4.0, -4.231, 1.0, mat2);
```

The result in both cases is that `mat1` is a transformation matrix assigned the value of multiplying the given translation by `mat2` in that order. `mat2` is passed by reference to both methods to improve efficiency.

To do post-translation, the code is similar:

```
vjMatrix mat1, mat2;
```

```

vjVec3 trans(4.0, -4.231, 1.0):
// Perform various transformations on mat2...
mat1.postTrans(mat2, trans);

```

(where `trans` is passed by reference) or

```

vjMatrix mat1, mat2;
// Perform various transformations on mat2...
mat1.postTrans(mat2, 4.0, -4.231, 1.0);

```

The result in both cases is that `mat1` is a transformation matrix assigned the value of multiplying `mat2` by the given translation in that order. `mat2` is passed by reference to both methods to improve efficiency. Note that for post-translation, the matrix argument is given before the translation argument(s) as a reminder that it comes first in the multiplication order.

Making a Scale Transformation Matrix

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

```

vjMatrix mat;
mat.makeScale(1.5, 1.5, 1.5);

```

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`.

As with the previous operations, more advanced changes can be made. Similar to the `preMult()` and `post#Mult()` methods described earlier, pre-scale and post-scale operations can be performed on a matrix. To do a pre-scale, the code would look similar to the following:

```

vjMatrix mat1, mat2;
// Perform various transformations on mat2...
mat1.preScale(1.5, 1.5, 1.5, mat2);

```

The result in both cases is that `mat1` is a transformation matrix assigned the value of multiplying the given scaling factors by `mat2` in that order. `mat2` is passed by reference to both methods to improve efficiency.

To do a post-scale, the code is similar:

```

vjMatrix mat1, mat2;
// Perform various transformations on mat2...
mat1.postScale(mat2, 1.5, 1.5, 1.5);

```

The result in both cases is that `mat1` is a transformation matrix assigned the value of multiplying `mat2` by the given scale factors in that order. `mat2` is passed by reference to both methods to improve efficiency. Note that for post-scaling, the matrix argument is given before the scale factor arguments as a reminder that it comes first in the multiplication order.

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 `vjMatrix` object representing arbitrary translation, rotation, and scaling transformations. To get the roll information, use the following:

```
float roll = mat.getRoll();
```

or

```
float z_rot = mat.getZRot();
```

The value return is in the range -180.0° to 180.0° . To get the pitch information, use:

```
float pitch = mat.getPitch();
```

or

```
float x_rot = mat.getXRot();
```

Again, the value return is in the range -180.0° to 180.0° . Finally, to get the yaw information, use:

```
float yaw = mat.getYaw();
```

or

```
float y_rot = mat.getYRot();
```

Though it may not need to be restated, the value return is in the range -180.0° to 180.0° .

Getting translations is even simpler because translations are collected into a single vector easily. There are two forms for getting the translation. The first takes three floats by reference:

```
float x, y, z;  
mat.getTrans(x, y, z);
```

After this, the translation in `mat` is stored in `x`, `y`, and `z`. The second form returns a copy of a `vjVec3` object:

```
vjVec3 trans;  
trans = mat.getTrans();
```

That is all there is to reading translations.

Converting to an OpenGL Performer Matrix

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

```
vjMatrix vj_mat;  
pfMatrix pf_mat;
```

```
// Perform operations on vj_mat...
```

```
pf_mat = vjGetPfMatrix(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 `vjMatrix`, do the following:

```
pfMatrix pf_mat;
```

```
vjMatrix vj_mat;
```

```
// Perform operations on pf_mat...
```

```
vj_mat = vjGetVjMatrix(pf_mat);
```

Here again, `pf_mat` is passed by reference for efficiency, and `vj_mat` gets a copy of a `vjMatrix`. Both of these functions are found in the header `Kernel/Pf/vjPfUtil.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

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 `vjMatrix` 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.

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 `vjMatrix` that acts as a full transformation matrix can be passed to OpenGL directly (more or less). For example, with a given `vjMatrix` object `mat` upon which painstaking transformations have been performed, the following can be done:

```
glMultMatrixf(mat.getFloatPtr());
```

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.

The `vjDeviceInterface` Helper Class

The concept of device interfaces in VR Juggler is one which often causes confusion for new users. Two object-oriented design patterns are combined by `vjDeviceInterface`: smart pointers and proxies. Within this section, we aim to explain VR Juggler device interfaces clearly and simply. We begin with a high-level description and then move right into using the class.

High-Level Description

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

The class `vjDeviceInterface` is designed to be a wrapper class around the proxies. Applications could use the proxy classes directly, but `vjDeviceInterface` and its subclasses simplify use of the proxy object they contain. Thus, typical VR Juggler application objects will have one or more device interface member variables.

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

With that background, we can move on to explain how `vjDeviceInterface` uses these concepts. First, know that `vjDeviceInterface` is a base class for all other device interface classes such as digital interfaces (wand buttons), position interfaces (wands, a tracked user's head), etc. In user code, there will be instances of objects such as `vjDigitalInterface`, `vjPosInterface`, `vjKeyboardInterface`, and the like. Once they are properly initialized, device interface objects (whatever their types may be) will act as smart pointers to the actual VR Juggler proxy objects they wrap.

All the subclasses of `vjDeviceInterface` encapsulate a pointer to a VR Juggler proxy object. (Remember that these proxy objects act as an intermediary between the application and an input device.) The subclasses also overload the dereference operator `->` which allows them to act as smart pointers. The dereference operator on a device interface object gives access to the object's hidden proxy pointer. With that access, the methods of the encapsulated proxy object can be invoked, usually to read data. The end result is that user applications get access to the proxy objects they need but through a simpler interface than using the proxies directly.

At this point, it is perfectly reasonable to wonder why VR Juggler uses a concept that requires all sorts of documentation and explanation. The extra effort is worth it because it allows VR Juggler 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 `vjDigitalInterface` gives exactly that information, and it quietly hides the messiness of dealing with that crazy mouse, its ugly driver, and its overly complex protocol.

Using `vjDeviceInterface`

VR Juggler applications do not usually use `vjDeviceInterface` directly. Instead, the subclasses mentioned above will be used. Within this section, we will refer to subclasses of `vjDeviceInterface` as “device interfaces”. The high-level description has already made use of this convention.

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

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

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

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

```
vjMatrix* head_mat;
head_mat = head->getData();
```

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

The Gory Details

What is truly amazing about VR Juggler 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 header files as a reference.

As mentioned, the class `vjDeviceInterface` is a base class for all the specific types of device interfaces such as positional interfaces, digital interfaces, and analog interfaces. This class maintains the name of the proxy and the proxy index, and it provides the all-important `init()` method, but it is up to the inheriting classes to handle the proxy pointer and to overload the dereference operator.

Subclasses of `vjDeviceInterface` are used to provide the wrapper to a specific type of proxy. They each contain a pointer to a proxy object of the same conceptual type (positional, digital, and so on). The way in which the dereference operator is overloaded can vary from class to class, but the end result is always the same: a pointer to the proxy is returned so that the calling code has access to that proxy.

The beauty of it all is that the proxy object being pointed to by the device interface can be changed without affecting

the execution of the user application. In other words, the proxies can be changed at run time to point to different *physical* devices. All the while, the user code is still using the smart pointer interface and getting data of some sort. This flexibility is one of the most important features of VR Juggler, and it is important to understand.

The `vjProxy` Helper Class

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

High-Level Description

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

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

Using `vjProxy`

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

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

The Gory Details

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

Part II. Application Programming

Table of Contents

4. Writing Applications	38
Application Review	38
Basic Application Information	38
Draw Manager-Specific Application Classes	38
Getting Input	38
How to Get Input	39
Where to Get Input	39
Tutorial: Getting Input	40
OpenGL Applications	41
OpenGL Drawing: vjGLApp::draw()	42
Tutorial: Drawing a Cube with OpenGL	42
Context-Specific Data	43
Using Context-Specific Data	45
Context-Specific Data Details	47
Tutorial: Drawing a Cube using OpenGL Display Lists	47
OpenGL Performer Applications	49
Scene Graph Initialization: vjPfApp::initScene()	49
Scene Graph Access: vjPfApp::getScene()	49
Tutorial: Loading a Model with OpenGL Performer	49
Other vjPfApp Methods	51
VTK Applications	53
5. Porting to VR Juggler from the CAVElib	54
The Initialize, Draw, and Frame Routines	54
In CAVElib	54
In VR Juggler	54
Getting Input from Devices	55
In CAVElib	55
In VR Juggler	55
Configuration	56
In CAVElib	56
In VR Juggler	56
Important Notes	56
Shared Memory	56
OpenGL Context-Specific Data	56
Source Code	56
The Form of a Basic CAVElib Program	57
The Form of a Basic VR Juggler Program	57
6. Porting to VR Juggler from GLUT	59
Window Creation and Management	59
The Initialize, Draw, and Frame Routines	59
In GLUT	59
In VR Juggler	59
Getting Input from Devices	60
In GLUT	60
In VR Juggler	60
Configuration	61
In GLUT	61
In VR Juggler	61
Important Notes	61
Shared Memory	61
OpenGL Context-Specific Data	61
Source Code	62
The Form of a Basic GLUT Program	62

The Form of a Basic VR Juggler Program 62

Chapter 4. Writing Applications

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

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 (`vjApp`). 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 `vjApp` or from a Draw Manager-specific application class that has `vjApp` as a superclass. For example:

```
class userApp : public vjApp
{
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 `vjApp`. In most cases, an application class is derived from a Draw Manager-specific application class. For example:

```
class userGApp : public vjGApp
{
public:
    init();
    preFrame();
    postFrame();

    draw();
}
```

This is an example of an OpenGL application. The application class (`userGApp`) has derived directly from the OpenGL Draw Manager-specific `vjGApp` 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 applications can use including positional, digital, and analog.

All applications share the same processes and concepts for acquiring input from devices. The main thing to remember about getting input in applications is that all VR Juggler applications receive input through device handles managed by `vjDeviceInterfaces`. There are `vjDeviceInterfaces` for each type of input data that VR Juggler can handle. There is one for positional input, one for analog, and so on. They all have very similar interfaces and behave exactly the same. (Refer to the section called “The `vjDeviceInterface Helper Class`” and the section called “The `vjProxy Helper Class`” for more information.)

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 vjApp
{
public:
    init();
    preFrame();
private:
    vjPosInterface mWand;
}
```

Note the declaration of the variable `mWand` of type `vjPosInterface`. 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.

```
...
vjMatrix wand_pos;
wand_pos = *(mWand->getData());
...
```

The above code shows an example of using the 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 “The `vjDeviceInterface Helper Class`” for more information about using `vjDeviceInterface`.

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.

This diagram looks complicated, but the key here is the `updateAllData()` call near the bottom of the diagram. This is where VR Juggler 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 `vjApp::preFrame()`, and
2. Any time spent in `vjApp::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

Table 4.1. Tutorial Overview

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	<ul style="list-style-type: none"> • <code>vjApp::init()</code> • <code>vjApp::preFrame()</code>
Directory	<code>\$VJ_BASE_DIR/share/samples/tutorials/simpleInput</code>
Files	<ul style="list-style-type: none"> • <code>simpleInput.h</code> • <code>simpleInput.cpp</code>

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 vjGApp
  {
  public:
    virtual void init();
5   virtual void preFrame();

  public:
    vjPosInterface      mWand;           // Positional interface for Wand position
    vjPosInterface      mHead;          // Positional interface for Head position
10   vjDigitalInterface mButton0;       // Digital interface for button 0
    vjDigitalInterface mButton1;       // Digital interface for button 1
  };

```

Initializing the Device Interfaces: `vjApp::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.

```

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

```

Examining the Device Data: `vjApp::preFrame()`

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

```

1 virtual void preFrame()
  {
    if ( mButton0->getData() ) ❶
    {
5     std::cout << "Button 0 pressed" << std::endl;
    }
    if( mButton1->getData() ) ❷
    {
10    std::cout << "Button 1 pressed" << std::endl;
    }

    std::cout << "Wand Buttons:" ❸
        << " 0:" << mButton0->getData()
        << " 1:" << mButton1->getData()
15    << std::endl;

    // -- Get Wand matrix --- //
    vjMatrix wand_matrix;
    wand_matrix = *(mWand->getData()); ❹
20    std::cout << "Wand pos: \n" << wand_matrix << std::endl;
  }

```

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

OpenGL Applications

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

In Figure 4.2, we see the functions added by the `vjGLApp` interface: `draw()`, `contextInit()`, and `contextPreDraw()`. These functions 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 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.

OpenGL Drawing: `vjGApp::draw()`

The most important (and visible) component of most OpenGL applications is the OpenGL drawing. The `vjGApp` 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 `vjGApp::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

Table 4.2. Tutorial Overview

Description	Simple OpenGL application that draws a cube in the environment
Objectives	Understand how the <code>draw()</code> member function in <code>vjGApp</code> works; create basic OpenGL-based VR Juggler applications
Member functions	<ul style="list-style-type: none"> • <code>vjApp::init()</code> • <code>vjGApp::draw()</code>
Directory	<code>\$VJ_BASE_DIR/share/samples/tutorials/simpleApp</code>
Files	<ul style="list-style-type: none"> • <code>simpleApp.h</code> • <code>simpleApp.cpp</code>

Class Declaration

The following application class is called `simpleApp`. It is derived from `vjGApp` 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 class simpleApp : public vjGApp
  {
  public:
    simpleApp();
5   virtual void init();
    virtual void draw();

  public:
    vjPosInterface mWand;
10   vjPosInterface mHead;
    vjDigitalInterface mButton0;
    vjDigitalInterface 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 void simpleApp::draw()
  {
    ...
5   vjMatrix box_offset;                                ❶
    box_offset.makeXYZEuler(-90,0,0);
    box_offset.setTrans(0.0,1.0f,0.0f);

    ...
10  glPushMatrix();
    // Push on offset
    glMultMatrixf(box_offset.getFloatPtr());          ❷
    ...
    drawCube();                                       ❸
15  glPopMatrix();
    ...
  }

```

- ❶ This creates a `vjMatrix` object that defines the offset of the cube in the virtual world.
- ❷ The new matrix is pushed onto the OpenGL modelview matrix stack.
- ❸ 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()` 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 intro#

duction 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.

```
class userApp : public vjGlApp
{
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:

```
class userApp : public vjG1App
{
public:
    draw();
public:
    vjG1ContextData<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 `vjG1ContextData<>` 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 (`vjG1ContextData<>`). 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()` 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 `vjGLApp::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 4.1. 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 application executes. This requires dynamic context data because the display list has to be regenerated every time the application changes the data set.

Consider also the following example. While running an application, the user requests to load a new model from a file. After the model data is loaded, it may be best to put the drawing functions into a fresh display list for rendering the model. In this case, `vjGLApp::contextInit()` cannot be used because it is only called when a new con#

text 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.

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

Table 4.3. Tutorial Overview

Description	Drawing a cube using a display list in the <code>draw()</code> member function
Objectives	Understand how to use context-specific data in an application
Member functions	<ul style="list-style-type: none"> • <code>vjApp::init()</code> • <code>vjGlfwApp::contextInit()</code> • <code>vjGlfwApp::draw()</code>
Directory	<code>\$VJ_BASE_DIR/share/samples/tutorials/contextApp</code>
Files	<ul style="list-style-type: none"> • <code>contextApp.h</code> • <code>contextApp.cpp</code>

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

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 4.1, 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 void contextApp::draw()
  {
    // Get Wand matrix
    vjMatrix wand_matrix;
5   wand_mat = *(mWand->getData());
    ...
    glPushMatrix();
    glPushMatrix();
    glMultMatrixf(wand_mat.getFloatPtr());
10  glCallList(*mCubeDlId);
    glPopMatrix();
    ...
    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 1.0, the supported scene graphs are OpenGL Performer from SGI 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 `vjPfApp`. Similar to `vjGLApp` presented in the previous section, `vjPfApp` derives from `vjApp`. `vjPfApp` extends `vjApp` by adding methods that deal with scene graph initialization and access. Figure 4.4 shows how `vjPfApp` fits into the class hierarchy of a Performer-based VR Juggler application.

Two of the methods added to the application interface by `vjPfApp` 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 `vjPfApp`). Additional methods will be discussed in this section, but in many cases the default implementations of these other methods may be used. A simple tutorial application will be provided to illustrate the concepts presented.

Scene Graph Initialization: `vjPfApp::initScene()`

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

Scene Graph Access: `vjPfApp::getScene()`

In order for Performer to render the application scene graph, it must get access to the scene graph root. The method `vjPfApp::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

Table 4.4. Tutorial Overview

Description	Simple OpenGL Performer 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	<ul style="list-style-type: none"> • <code>vjPfApp::initScene()</code> • <code>vjPfApp::getScene()</code>
Directory	<code>\$VJ_BASE_DIR/share/samples/tutorials/simplePf</code>
Files	<ul style="list-style-type: none"> • <code>simplePfApp.h</code> • <code>simplePfApp.cpp</code>

Class Declaration

The following application class is called `simplePfApp`. It is derived from `vjPfApp` 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 vjPfApp
  {
  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);

```

①

②

```

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());           ❸

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

```

- ❶ First, the root node is constructed as a `pfGroup` object.
- ❷ Next, some steps are taken to create a light source for the application.
- ❸ 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()`.)
- ❹ 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 `vjPfApp` Methods

Besides the two methods discussed so far, there are several other methods in `vjPfApp` that extend the basic `vjApp` 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()`).

VTK Applications

Chapter 5. Porting to VR Juggler from the CAVElib™

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

The Initialize, Draw, and Frame Routines

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

In CAVElib™

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

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

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 `vjGApp`. 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 vjGApp
{
    ...
};
```

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 CAVElib™ software and VR Juggler.

In CAVElib™

To get tracking information, the following functions are used:

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

For button input, the following macros are used:

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

In VR Juggler

To get device input, use the classes derived from `vjDeviceInterface`. They include the following:

- `vjPosInterface` (for trackers and other positional devices)
- `vjDigitalInterface` (for buttons and other on/off devices)
- `vjAnalogInterface` (for potentiometers and other multi-range data devices)

For more information about the VR Juggler device interfaces, refer to Chapter 3, *Helper Classes*. 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 CAVElib™ software is very different. The differences are too numerous to list here, but we give a brief overview and a pointer to the documentation that explains configuration of VR Juggler.

In CAVElib™

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

In VR Juggler

Configuration of VR Juggler is much more powerful and flexible than what is used by the CAVElib™ software. As a result, it is also more complex. All configurable parameters could be in one or more files with any names desired. VR Juggler comes with example configuration files that may be found in the directory `$VJ_BASE_DIR/share/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 VjControl book 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 CAVElib™ software, VR Juggler does not have to manage shared memory with other VR Juggler instances. Thus, when writing a VR Juggler application, memory can be created as in a normal, single-threaded C or C++ application.

OpenGL Context-Specific Data

As a result of the shared memory model described above, VR Juggler has different requirements for context-specific data than the CAVElib™ software. Information such as display lists and texture objects must be managed using context-specific data. A *display context* is the location to which OpenGL rendering commands draw. Compiled OpenGL commands such as display lists do not get shared across multiple contexts (or windows), and thus, they must be initialized once per display context. In a VR Juggler application, these OpenGL initializations must be placed in `vjGLApp::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 CAVElib™ concepts map to VR Juggler.

The Form of a Basic CAVElib™ 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);
15   while (!getbutton(ESCKEY))
    {
        app_compute();
    }
    CAVEExit();
20 }
```

The Form of a Basic VR Juggler Program

```
1 class MyApplication : public vjGlApp
  {
  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
    vjKernel* kernel = vjKernel::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);

    // hit CTRL-C to exit.
    while (1)
    {
35     // Juggler executes within other threads, so sleep here.
        usleep( 100000 );
    }
  }
```

}

Chapter 6. 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. Console text initialization is done during the first run of the function set with `glutDisplayFunc()` (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 `vjGApp`. 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 vjGApp
{
    ...
};
```

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 classes derived from `vjDeviceInterface`. They include the following:

- `vjPosInterface` (for trackers and other positional devices)
- `vjDigitalInterface` (for buttons and other on/off devices)
- `vjAnalogInterface` (for potentiometers and other multi-range data devices)

For more information about the VR Juggler device interfaces, refer to Chapter 3, *Helper Classes*. 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/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 *VjControl* book 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 `vjGlApp::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 configuration file
     * 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
     *       vjGApp::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( OnMouseClicked );
    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(1)
     * (see below)
     */
45   glutMainLoop();
  }
```

The Form of a Basic VR Juggler Program

```
1 class MyApplication : public vjGApp
  {
```

```
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
    vjKernel* kernel = vjKernel::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);

    // hit CTRL-C to exit.
    while (1)
    {
35  // Juggler executes within other threads, so sleep here.
        usleep( 100000 );
    }
}
```

Part III. Advanced Topics

Table of Contents

7. System Interaction	66
8. Multi-threading	67
Techniques	67
Helper Classes	67
Using the vjThread Interface	67
Using the vjBaseThreadFunctor Interface	72
Using the vjSemaphore Interface	75
Using the vjMutex Interface	76
9. Run-Time Reconfiguration	79
How Run-Time Reconfiguration Works	79
Reasons to Use Run-Time Reconfiguration	79
Using Run-Time Reconfiguration in an Application	79
Application Tutorial	81
10. Adding Device Drivers to VR Juggler	82
In-Depth Driver Guide	82
Implementing the Device Driver	82
Register the Device Driver with VR Juggler	85
Device Driver Configuration	85
Configuration Files	85
Writing Code that Accepts the Configuration	86
Example Code	86

Chapter 7. 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, run-time reconfiguration, and device driver authoring. 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 8. 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 `vjApp::intraFrame()` application object member function
2. Use triple-buffered data
3. Use separate threads with the `vjThread` class

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 thread API. This section presents all the helper classes available for multi-threaded VR Juggler application programming. We begin with the thread API and then move on to the synchronization primitive API.

Using the `vjThread` Interface

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.

Recommended Reading

It is assumed that readers already know the basics of multi-threaded programming including the definition of *thread of control*. What is described here is how to use the VR Juggler thread interface, `vjThread`, not how to write multi-threaded software. For that reason, it is recommended that readers be familiar with the following publications before continuing:

- *Pthreads Programming* [<http://www.oreilly.com/catalog/pthread/>] by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell.

- The `sproc(2)` manual page on IRIX or on SGI's technical publications site [http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/p_man/cat2/standard/nsproc.z&srch=sproc].
- The `pthread(3)` manual page for your operating system. The `pthread` functions are part of a POSIX standard and will be the same across platforms.

High-Level Description

The threading interface in VR Juggler is modeled after the POSIX thread specification of POSIX.1b (formerly POSIX.4). The main difference is that VR Juggler'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 `vjThread` 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 `vjThread` constructor is called. That is, when instantiating a `vjThread` 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 `vjThreads`. Do not make any assumptions about when the thread will begin running. It may happen before or after the constructor returns the `vjThread` 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 `vjThread` constructor.

Once a `vjThread` 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.

Using vjThread

Use of `vjThread` is intended to be easy. Multi-threaded programming has enough complications without having a difficult API as well. In almost all cases, thread creation can be done in a single step, executed one of two ways:

1. Pass a function pointer to the `vjThread` constructor along with any argument that should be passed to the function when the thread is created
2. Pass a *functor* to the `vjThread` constructor

The second appears easier, but to create the functor, an argument to the function executed by the thread may still have to be passed. The presence of the argument depends on the specific function being run by the thread. In addition to the function pointer or functor, parameters such as the priority and the stack size may be passed to the `vjThread` constructor, but the defaults for the constructor are quite reasonable.

A minor issue with creating a `vjThread` is the concept of functors. The topic of functors will be put off until another section. For now, just think of them as wrappers around function pointers.

Before writing code that uses `vjThreads`, make sure that the header file `Threads/vjThread.h` is included. Never include the platform-specific headers such as `Threads/vjThreadPosix.h`. The single file `Threads/vjThread.h` is all that is required.

Creating vjThreads

The following example illustrates how to create a thread that will execute a function called `run()` that takes no arguments. The prototype for `run()` is:

```
void run(void* args);
```

This will be the same across all platforms. The thread creation code is then:

```
vjThread* thread;  
thread = new vjThread(run);
```

At this point, a newly spawned thread is executing the code in `run()`. It is advisable to hang onto the variable `thread` so that the thread may be controlled as necessary.

That was pretty easy. What if you want to pass one or more arguments to `run()` so that its behavior can be modified based on some variables? Not surprisingly, that is fairly easy too. As mentioned above, if there is more than one argument to pass to the thread function, they will have to be collected into a struct, and pointer to that struct will have to be passed. A common way to do this is as follows:

```
struct ThreadArgs  
{  
    int id;  
    char name[40];  
    // And so on...  
};  
  
void someFunc ()  
{  
    // Other code ...  
  
    ThreadArgs* args;  
    vjThread* thread;  
  
    args = new ThreadArgs();  
  
    // Fill in the elements of args ...  
  
    thread = new vjThread(run, (void*) args);  
}
```

When creating a single thread, this works beautifully. If multiple threads are needed, all taking the same type of argument, there must be a separate argument structure instance for each one. A bunch of pointers can be declared, or the same pointer can be reused over and over. The address passed to each thread will be unique either way. Using this method requires that the argument memory be released before the thread exits, of course.

Waiting for a Thread to Complete

Once we have a thread running, it is often useful to synchronize another thread so that its execution halts until the running thread has completed. This is called “joining threads”. The following example illustrates how this can be done:

```
vjThread* thread;  
thread = new vjThread(run);  
  
// Do other things while the thread is going ...  
  
thread->join();  
  
// Now that the thread is done, continue.
```

Here, the creator of thread can be another `vjThread`, or it can be the main thread of execution (though it will not be that way in a VR Juggler application). In other words, any thread can create more threads and control them. What happens in this example is that thread is created and begins running. Meanwhile, the creator thread continues to do some more work and then must wait for `thread` to finish its work before continuing. It calls the `join()` method, a blocking call, and it will not return until `thread` has completed.

While it is not demonstrated here, the `join()` method can take a single argument of type `void**`. It is a pointer to a pointer where the exit status of the joined thread is stored. The operating system fills the pointed to pointer with the exit status when the thread exits.

It is very important to note that in VR Juggler 1.0, the implementation of thread joining is incomplete for IRIX SPROC threads. The next major release of VR Juggler will have this corrected.

Suspending and Resuming a Thread's Execution

Sometimes, it may be necessary to suspend the execution of a running thread and resume it again later. There are two methods in the `vjThread` interface that do just this. Assuming that there is already a running thread pointed to by the object `thread`, it can be suspended as follows:

```
thread->suspend();
```

Resuming execution of the suspended thread is just as easy:

```
thread->resume();
```

On successful completion, both methods return 0. If the operation could not be performed for some reason, -1 is returned to indicate error status.

Getting and Setting a Thread's Priority

Changing the priority of a thread tells the underlying operating system how important a thread is and gives it hints about how to schedule the threads. If no value for the priority is given to the constructor, all `vjThreads` are created with the default priority for all threads. Values higher than 0 for the priority request a higher priority when the thread is created.

Besides being able to set the priority when the thread is created, it is possible to query and to adjust the priority of a running thread. Assuming that there is already a running thread pointed to by the object `thread`, its priority can be requested as follows:

```
int prio;
thread->getPrio(&prio);
```

The thread's priority is stored in `prio` and returned via the pointer passed to the `getPrio()` method. Setting that thread's priority is also easy:

```
int prio;
// Assign some priority value to prio ...
thread->setPrio(prio);
```

On successful completion, both methods return 0. If the operation could not be performed for some reason, -1 is returned to indicate error status.

Sending Signals to a Thread

On UNIX-based systems, a signal is sent to a process using the `kill(2)` system call. With POSIX threads, signals are sent using `pthread_kill(3)`. VR Juggler's thread interface implements these ideas using a `kill()` method. There are two ways to call this method: with an argument naming the signal to be delivered to the thread or without an argument which cancels the thread's execution. The first of these is described in this section, and the second is described in the next section.

A problem does arise here, unfortunately. Signals are not supported on all operating systems (notably, Win32). The interface is consistent, but code written on IRIX will not compile on Win32 if, for example, it sends a `SIGHUP` to a thread. An improved thread interface is being designed to overcome problems such as this one. For now, we describe this part of the interface as though it is supported completely on all platforms.

As usual, assume there is a running thread, a pointer to which is stored in `thread`. To send it a signal (`SIGINT`, for example), use the following:

```
thread->kill(SIGINT);
```

The signal will be delivered to the thread by the operating system, and the thread is expected to handle it properly. This version of the `kill()` method returns 0 if the signal is sent successfully. Otherwise, -1 is returned to indicate that an error occurred.

Canceling a Thread's Execution

As described in the previous section, using the `kill()` method with no argument cancels the execution of the thread. When using POSIX threads, this is actually implemented using `pthread_cancel(3)`. On IRIX with SPROC threads, a `SIGKILL` is sent to the thread to end its execution forcibly. The syntax for using this method is basically the same as in the previous section, but it is repeated to make that clear. Again assuming that there is a running thread with a pointer to its `vjThread` object stored in `thread`, use the following:

```
thread->kill();
```

Unlike the syntax used to send a signal to a thread, this version of `kill()` does not have a return value.

Users of POSIX threads may be wondering if the `vjThread` API provides a way to set cancellation points in the code. Unfortunately, it does not at this time. Extending the interface in this way is being considered, but cancellation points do not have meaning with all thread implementations.

Requesting the Current Thread's Identifier

Lastly, it is common to request the currently running thread's identifier. This only makes sense when called from a point on that thread's flow of execution. (In POSIX threads, this is the notion of "self". For IRIX SPROC threads, this means getting the process ID.) The `vjThread` API provides a static method that can be called at any time in the thread that is currently running. It returns a pointer to a `vjBaseThread` (the basic type from which `vjThread` inherits its interface). The syntax is as follows:

```
vjBaseThread* my_id;
my_id = vjThread::self();
```

The returned pointer can then be used to perform all of the previously described operations on the current thread.

The Gory Details

The current threading implementation in VR Juggler is a little difficult to understand. The code is not complicated at all, but because all platform-specific implementations are referred to as `vjThreads`, the details can get lost in the

shuffle. To begin, the current list of platform-specific thread implementation wrapper classes are:

- `vjThreadSGI`: A wrapper around IRIX SPROC threads (refer to the `sproc(2)` manual page for more information)
- `vjThreadPosix`: A wrapper around POSIX threads (both Draft 4 and Draft 10 of the standard are supported)
- `vjThreadWin32`: A wrapper around Win32 threads

The interface itself is defined in `vjBaseThread`, and all of the above classes inherit from that class.

The threading implementation used is chosen when VR Juggler is compiled. To use a certain type of thread system, be sure that the version of VR Juggler in use was compiled with the type of threads desired. When the VR Juggler build is configured, preprocessor `#define` statements are made in `vjDefines.h` that describe the threading system to use. Based on that, the header file `Threads/vjThread.h` makes several typedefs that set up one of the platform-specific thread implementations to act as the `vjThread` interface. For example, if compiling on Win32, the class `vjThreadWin32` is typedef'd to be `vjThread`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

The current implementation is modeled after the POSIX thread API for the most part. When designing it, we approached it with the idea that having a more complete API was more important than having a “lowest-common-denominator” API. That is, just because not all threading implementations support a specific feature does not mean that the API should suffer by not having that feature. Whether this was a good approach or not is an open debate.

As mentioned in the previous section, the next major release of VR Juggler (2.0 as of this writing) dramatically changes the way the threading subsystem in VR Juggler is implemented. We have added a wrapper around Netscape Portable Runtime [<http://www.mozilla.org/projects/nspr/index.html>] (NSPR) threads. We have removed the Win32-specific threads because NSPR already supports that implementation. Further implementations may be removed in favor of using what NSPR offers. Doing this will offload much of our efforts onto the NSPR. Most of what has been described in the `vjThread` interface has remained consistent after this change. NSPR threads do not support all the features we have, however, because they took the lowest-common-denominator approach. As with all technology, there is a trade-off in relieving some of our work load by using an existing cross-platform thread implementation: our interface becomes limited to what features that implementation provides. It remains to be seen exactly how much of VR Juggler's threading subsystem will be removed, and those programmers who choose to use it should be careful to watch the mailing lists for discussions and announcements about changes.

Using the `vjBaseThreadFunctor` Interface

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 VR Juggler, functors can be used as the code executed by a thread (refer to the section called “Using the `vjThread` Interface” for more detail on the topic of `vjThreads`). This section describes how to use functors for exactly that purpose.

High-Level Description

As mentioned, a functor is used in VR Juggler with `vjThreads`. VR Juggler's threads can execute two types of functions: normal C/C++ functions and class methods or functors. The former was described in the section about using `vjThreads`, and the latter is described here. The use of functors is given more attention because the concept may be foreign to some programmers. Those who already know about functors can skip this short description and go straight to the section on using functors.

In VR Juggler, 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 im#

plement 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 `vjThreads`, 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 `vjThread` 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 VR Juggler functor, that is all handled behind the scenes so that passing a non-static member function is straightforward.

Before getting into specifics, there is a header file that must be included to use VR juggler thread functors. In this case, the header is `Threads/vjThreadFunctor.h`. Within this header, `vjBaseThreadFunctor` is declared as an abstract base class. It has two subclasses implementing its interface: `vjThreadMemberFunctor` and `vjThreadNonMemberFunctor`. Both of these subclasses will be discussed in turn next.

vjThreadMemberFunctor

This implementation of `vjBaseThreadFunctor` is for all functions that fall into the rather elite category of being non-static class member functions. To be more specific, those member functions (heretofore referred to as “methods”) must have the following prototype:

```
void methodName(void* arg);
```

Those readers with experience in multi-threaded programming will recognize this prototype instantly. It is no different than that used in common threading implementations. Constructing the functor to use this method, however, is quite different than what readers may have seen before.

Say there is a class `MyObject` with a method `run()` having the appropriate prototype that will be executed by a `vjThread` object. In this case, `run()` takes an argument that is a pointer to a pre-defined type `thread_args_t`. Also assume that there is an instance of `MyObject` pointed to by the variable `my_obj`. The following code creates the `vjThreadMemberFunctor` object that will encapsulate the method:

```
vjThreadMemberFunctor<MyObject>* my_functor;  
thread_args_t* args;  
vjThread* thread;  
  
args = new thread_args_t();  
  
// Fill in the arguments to be passed to the thread...  
  
my_functor = new vjThreadMemberFunctor<MyObject>(my_obj, &MyObject::run, (void*) args);  
thread = new vjThread(my_functor);
```

The important thing to note in this example is that `vjThreadMemberFunctor` is a template class. When creating the functor instance, the class must be specified as the template parameter. (If you do not understand this syntax, take a look at a C++ book that covers the current C++ standard.) Also note that when creating the new `vjThread` object, the argument structure is not passed to the constructor. The argument to `run()` is packaged up with the function in the functor object `my_functor`. Once this code has executed, a new thread is spawned that will run the `run()` method given the provided argument structure.

The `vjBaseThreadFunc`tor interface defines an extra method `setArg()` that allows the function's argument to be set after the functor object is created. The argument to the constructor providing the function's argument is optional and will default to `NULL` if not specified. At a later time, should there be an argument to provide, the following can be used:

```
thread_args_t* args = new thread_args_t();

// Fill in args ...

my_functor->setArg(args);
```

This assumes that there is already a functor object instantiated called `my_functor`. Alternatively, `setArg()` could be used to remove a previously defined argument by passing `NULL`.

The last thing to note is that a lot of memory is being allocated dynamically in the example code. Be careful to deallocate the memory when it is no longer needed.

vjThreadNonMemberFunctor

This implementation of `vjBaseThreadFunc`tor is the complement of the set of functions contained by `vjThreadMemberFunc`tor. It is used for normal C/C++ functions and for static class member functions. There is nothing terribly interesting about this class, and its use is straightforward. The following example, an adaptation of that presented in the previous section, shows how to use this interface rather than passing a function pointer and an argument to the `vjThread` constructor. In this case, assume that the function `run()` is appropriately defined for use here.

```
vjThreadNonMemberFunc
```

tor* my_functor;
thread_args_t* args;
vjThread* thread;

args = new thread_args_t();

// Fill in the arguments to be passed to the thread...

my_functor = new vjThreadNonMemberFunctor(run, (void*) args);
thread = new vjThread(my_functor);

That is all there is to it. Programmers end up doing more work than if they had just passed the function pointer and the associated argument to the `vjThread` constructor directly, but the `vjThread` constructor is relieved of some work. (The reason for this is described below.) Thus, either way is equally efficient, and what you use is up to you.

The Gory Details

The magic behind these functors is done by overloading `operator()` for `vjBaseThreadFunc`tor objects. Both implementations of the interface store the function pointer (and optional argument pointer), and when `vjBaseThreadFunc`tor::`operator()` is invoked, they call the function and pass the argument if there is one. There is a little more magic with the `vjThreadMemberFunc`tor, however, that allows it to work with the non-static methods of a given class.

The class `vjThreadMemberFunc`tor works its extra-special magic through the use of a template and one of C++'s dustier operators, `::*`. This operator is used to point to a member of a class. In this case, it points to the method that will be executed by the thread. When used in conjunction with the provided class instance (the `this` pointer), the non-static method can be invoked by the functor.

One interesting thing to note about `vjThreads` is that they deal only in functors. More specifically, they deal only with objects that subsume to `vjBaseThreadFunc`tor. If a function pointer and its argument are passed directly to the `vjThread` constructor, a `vjThreadNonMemberFunc`tor object is created to package those arguments.

That new functor is then used internally by the thread. Thus, whether you choose to create a non-member functor or to pass the function pointer and associated argument, the same code will be executed.

Using the `vjSemaphore` 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 VR Juggler. 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.

High-Level Description

As with threads, a cross-platform abstraction layer has been written into VR Juggler 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. The interface does include methods for read/write semaphores, but as of this writing, that part of the interface is not complete. Because of that, the use section does not cover that part of the interface. When the implementation is complete, this section will be expanded.

As always, there is a header file that must be included to use `vjSemaphore`. This time around, the file is `Sync/vjSemaphore.h`. Do not include any of the platform-specific implementation files. That is all handled appropriately within `Sync/vjSemaphore.h`.

Creating a `vjSemaphore`

When creating a `vjSemaphore` object, give the initial value that represents the number of resources being controlled by the semaphore. If no value is given, the default is 1 which of course gives a binary semaphore. Binary semaphores are better known as mutexes (see the section called “Using the `vjMutex` Interface” for more information about mutex use in VR Juggler). An example of creating a simple semaphore to control access to five resources is as follows:

```
vjSemaphore sema(5);
```

This creates a semaphore capable of controlling concurrent access to five resources. At some point, if there is a need to change the number of resources, a method called `reset()` is provided. Pass the new number of resources, and the semaphore object is updated appropriately:

```
sema.reset(4);
```

The semaphore `sema` now controls access to only four resources.

Locking a Semaphore

When a thread needs to acquire access to shared data, it locks a semaphore. In the `vjSemaphore` interface, this is accomplished using the `acquire()` method:

```
sema.acquire();
```

As expected, `acquire()` is a blocking call, so if the semaphore's value is less than or equal to 0, the thread requesting the lock will block until the semaphore's value is greater than 0. Note that the return value of `acquire()` is a little different than most calls. If the lock is acquired, 1 is returned. If the attempt to lock the semaphore fails for some reason, -1 is returned. The newer API associated with VR Juggler 2.0 resolves this inconsistency.

Releasing a Locked Semaphore

Finally, when access to the critical section is complete, the semaphore is released using the `release()` method:

```
sema.release();
```

If the locked semaphore is released successfully, 0 is returned. Otherwise, -1 is returned.

The Gory Details

Those who have read the Gory Details section for `vjThreads` will find this section very familiar. As with `vjThreads`, there are several platform-specific semaphore implementation wrapper classes:

- `vjSemaphoreSGI`: A wrapper around IRIX shared-arena semaphores (refer to the `usnewsema(3P)` and related manual pages for more information)
- `vjSemaphorePosix`: A wrapper around POSIX real-time semaphores (POSIX.1b, formerly POSIX.4)
- `vjSemaphoreWin32`: A wrapper around Win32 semaphores

Unlike `vjThread`, however, there is no base interface from which these implementations inherit. Performance decreases caused by virtual functions are avoided this way.

The semaphore implementation used is chosen when VR Juggler is compiled and will always match the thread implementation being used. When the VR Juggler build is configured, preprocessor `#define` statements are made in `vjDefines.h` that describe the threading system and thus the semaphores to use. Based on that, the header file `Sync/vjSemaphore.h` makes several typedefs that set up one of the platform-specific implementations to act as the `vjSemaphore` interface. For example, if compiling on Linux, the class `vjSemaphorePosix` is typedef'd to `vjSemaphore`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

Using the `vjMutex` Interface

In addition to cross-platform semaphores, VR Juggler 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 `vjMutex` interface provided with and used by VR Juggler.

High-Level Description

The cross-platform mutex abstraction in VR Juggler is critical for synchronizing access to shared data. Those who have read the section on `vjSemaphore` will find this section very, very familiar. The interface for `vjMutex` is a subset of that for `vjSemaphore` since mutexes are binary semaphores. They can be locked and unlocked. That is all there is to know. The `vjMutex` 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.

The header file to include for using `vjMutex` is `Sync/vjMutex.h`. As with other classes discussed in this chapter, it is important not to include the platform-specific header files.

Creating a `vjMutex`

When creating a `vjMutex` object, there are no special parameters to pass or considerations to be made. An example of creating a mutex is as follows:

```
vjMutex mutex;
```

There is nothing more to say this time.

Locking a Mutex

When a thread needs to acquire access to shared data, it can lock a mutex. In the `vjMutex` interface, this is accomplished using the `acquire()` method:

```
mutex.acquire();
```

As expected, `acquire()` is a blocking call, so if the mutex is already locked by another thread, the thread requesting the lock will block until the mutex is released by the other thread. Note that the return value of `acquire()` is a little different than most calls. If the lock is acquired, 1 is returned. If the attempt to lock the semaphore fails for some reason, -1 is returned. The newer API associated with VR Juggler 2.0 resolves this inconsistency.

Attempting to Lock a Mutex

If there is a need to lock a mutex only when the call would *not* block, a method is provided to do this. It is called `tryAcquire()`, and it will not block if the mutex is already locked. It works as follows:

```
mutex.tryAcquire();
```

If the mutex is locked, 1 is returned. Otherwise, 0 is returned. The call does not block.

Testing the State of a Mutex

In addition to conditional locking, the state of a mutex can be tested to see if it is locked or unlocked. This is done using the `test()` method as follows:

```
int state = mutex.test();
```

If the mutex is *not* locked, 0 is returned. Otherwise, 1 is returned.

Releasing a Locked Mutex

When access to the critical section is complete, a locked mutex is released using the `release()` method:

```
sema.release();
```

If the locked mutex is released successfully, 0 is returned. Otherwise, -1 is returned.

The Gory Details

Those who have read the Gory Details sections for `vjThreads` or for `vjSemaphores` will find this last section very familiar (and probably uninteresting at this point). As with `vjThreads` and `vjSemaphores`, there are several platform-specific mutex implementation wrapper classes:

- `vjMutexSGI`: A wrapper around IRIX shared-arena mutexes (refer to the `usnewlock(3P)` and related manual pages for more information)
- `vjMutexPosix`: A wrapper around POSIX real-time mutexes (POSIX.1b, formerly POSIX.4)
- `vjMutexWin32`: A wrapper around Win32 mutexes

Similar to `vjSemaphore`, there is no base interface from which these implementations inherit. Performance issues caused by virtual functions are avoided by doing this.

The mutex implementation used is chosen when VR Juggler is compiled and will always match the thread implementation being used. When the VR Juggler build is configured, preprocessor `#define` statements are made in `vjDefines.h` that describe the threading system and thus the mutexes to use. Based on that, the header file `Sync/vjMutex.h` makes several typedefs that set up one of the platform-specific implementations to act as the `vjMutex` interface. For example, if compiling on Solaris, the class `vjMutexPosix` is typedef'd to be `vjMutex`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

Chapter 9. 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.

1. Define application chunk descriptions (ChunkDescs)

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 ConfigChunks that will contain it. This essentially means creating a file containing one or more ChunkDescs. To understand this better, consider the following example. One might define an “Object” ChunkDesc in an application. The ChunkDesc 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 ChunkDescs are read by the application. One way is to load the ChunkDesc file explicitly (described below), but the simplest way is to include the custom ChunkDesc file from one of the configuration files the application loads at startup. Instructions for editing ChunkDesc files and creating new kinds of ChunkDescs are included in the *VjControl User's Guide*.

2. Implement dynamic reconfiguration interface

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

- a.
- ```
virtual bool configCanHandle(vjConfigChunk* chunk);
```

This function should simply return a Boolean (true or false) depending on whether this object knows how to deal with the ConfigChunk passed to it. If this Chunk uses an application-custom ChunkDescs, this

should return true. For example:

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

if (!vjstrcasecmp(s, "my_custom_chunk_type"))
{
 return true;
}
```

b.

```
virtual bool configAdd(vjConfigChunk* chunk);
```

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

When `configAdd()` is called, the application should look at the chunk passed to it and decide what to do. This might involve creating a new object, changing the configuration of an already 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.

c.

```
virtual bool configRemove(vjConfigChunk* chunk);
```

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

### 3. Processing ConfigChunks

When an application receives a ConfigChunk to process via `configAdd()` or `configRemove()`, it needs to retrieve the data in that Chunk in order to decide what to do. ConfigChunks 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 ConfigChunk API.

- ```
std::string& vjConfigChunk::getType();
```

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

- ```
vjVarValue& vjConfigChunk::getProperty(std::string& property_token,
 int num);
```

This is the key method for getting the information contained in a ConfigChunk. 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 for this method needs some explanation. `vjVarValue` is a placeholder class designed to be cast safely to one of a variety of data types. Basically, the return value of `getProperty()` is cast to whichever type the caller expects to receive. It is assumed that the caller knows the types of values stored in a given property.

`vjVarValue` tries to coax the data in a variety of ways for convenience. For example, the `VarValue` returned from an integer, float, or boolean property can be safely cast to a string or `char*`. Booleans can be cast to integers and vice versa. The following code fragment gives a few examples of this usage:

```
std::string s1 = (std::string) chunk->getProperty("name", 0);
char* s2 = (char*) chunk->getProperty("name");
// NOTE: the cast allocates new memory for s2, which you are
// responsible for deleting.

bool b = (bool) chunk->getProperty("enabled");
std::string s3 = (std::string) chunk->getProperty("enabled");
// s3 will be one of the strings "True" or "False"

vjConfigChunk* ch = (vjConfigChunk*) chunk->getProperty("embedded_chunk", 2);
// ch is a copy of the embedded chunk. Once again, the caller
// of getProperty() is responsible for freeing this memory.
```

- ```
const int vjConfigChunk::getNum(const std::string& property_token);
```

Sometimes properties of a `ConfigChunk` 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 `ConfigChunk` API, refer to the *VR Juggler Programmer's Reference*.

4. Loading and saving configurations

Application Tutorial

Chapter 10. Adding Device Drivers to VR Juggler

In this final chapter, we explain how to add device drivers to VR Juggler. We begin with a detailed description of device driver conventions in VR Juggler and how the drivers fit into the VR Juggler Input Manager. We then explain how device drivers are configured. The chapter concludes with example code showing a very simple driver that reads button presses.

In-Depth Driver Guide

All device drivers in VR Juggler must derive from one or more of the following classes:

- `vjInput` (base case of all device drivers)
- `vjDigital`, `vjSimDigital`
- `vjAnalog`, `vjSimAnalog`
- `vjPosition`, `vjSimPosition`
- `vjGlove`, `vjSimDigitalGlove`, `vjGloveGesture`

For example, to make a driver that registers button presses, derive from `vjDigital`:

```
class MyNewButtonDevice : public vjDigital
```

Supposed that a joystick driver supporting buttons and movement is needed. In this case, an additional component, this one for analog input, is needed for the X and Y axes. Since the device is both digital and analog, its class must derive from both `vjDigital` and `vjAnalog` using C++ multiple inheritance:

```
class MyNewJoystickDevice : public vjDigital, public vjAnalog
```

Note

To use the joystick in place of a tracker, it should derive instead from `vjPosition`. This way, you can replace real trackers with your joystick “pseudo tracker”. The main idea is that to be able to replace one device with another, the alternate device class must derive from the same base classes as the original device.

Implementing the Device Driver

Using basic class declaration for `MyNewButtonDevice` from above, we will proceed with the implementation of the driver class. First, there are six member functions that must be implemented:

1.

```
virtual int startSampling();
```

Within this function, a new thread is started. This thread is used to sample the data from the device. The thread creation step may look something like the following:

```

vjThreadMemberFunctor<MyNewButtonDevice>* functor =
    new vjThreadMemberFunctor<MyNewButtonDevice>(this,
                                                    &MyNewButtonDevice::sampleFunction,
                                                    NULL);
mThread = new vjThread(functor);

```

The above creates a thread that will execute `MyNewButtonDevice::sampleFunction()`, a non-static member function in the class `MyNewButtonDevice`. The implementation of that method would be similar to the following in most cases

```

void MyNewButtonDevice::sampleFunction (void* arg)
{
    // Keep working until someone kills mThread.
    while ( 1 )
    {
        this->sample();
    }
}

```

The thread can be tested for validity using the method `vjBaseThread::valid()`. For more information on VR Juggler threads, refer to the section called “Using the `vjThread` Interface”.

2. `virtual int stopSampling();`

The job of this function is to kill the thread created in `startSampling()`. Again, refer to the section called “Using the `vjThread` Interface” for more information on the use of VR Juggler threads.

3. `virtual int sample();`

This method reads data from the device and stores it for later use by `getDigitalData()`. Note that `MyNewButtonDevice::sampleFunction()`, defined above, invokes this method.

VR Juggler devices typically use triple-buffered data management. This is done to ensure that data is not being written into a buffer when the Input Manager is trying to read the most recent value. The `vjInput` class defines three variables to help programmers keep track of which buffer is in use at any given time: `vjInput::current`, `vjInput::valid`, and `vjInput::progress`. The sampled data would be read into a three-element array of the correct type (this is driver-specific). When writing the freshly sampled data into the array, use `vjInput::progress`:

```

mSampledDigitalData[vjInput::progress] = sampled_digital_value;

```

4. `virtual void updateData();`

Triple-buffered device drivers use this method to swap the data indices. The member function is usually implemented as follows:

```

void MyNewButtonDevice::updateData()
{
    vjGuard<vjMutex> updateGuard(lock);

    // Copy the valid data to the current data so that both are valid
    mSampledDigitalData[current] = mSampledDigitalData[valid];
}

```

```

    // swap the indices for the tri-buffer pointers
    vjInput::swapCurrentIndexes();
}

```

Note the use of a `vjGuard<>` object to synchronize access to the `mSampledDigitalData` array. This is needed because the sampling and the reading are occurring in separate threads, but both threads need access to `mSampledDigitalData`.

5.

```

static std::string getChunkType();

```

In the `getChunkType()` function, the *chunk type* of the device must be returned. Its name must be as it appears in the chunk description file for the driver. For example, the implementation for the simple button driver would appear as:

```

std::string MyNewButtonDevice::getChunkType()
{
    return std::string("MyNewButtonDevice");
}

```

At this time, it is useful to point out that every VR Juggler device needs a *chunk type* associated with it. A chunk type is similar to a struct in C or C++. The data structure is defined in a chunk description file (which usually has the extension `.desc`). Once defined, the type for a new driver can be used in VR Juggler configuration files.

6.

```

virtual int getDigitalData(int devNum = 0);

```

The VR Juggler Input Manager uses this method to read digital data sampled by the driver. This is when the triple-buffered data scheme becomes especially valuable. To provide the Input Manager with the most up-to-date sample, use `vjInput::current` as the index, as shown below:

```

int MyNewJoystickDevice::getDigitalData(int devNum)
{
    return mSampledDigitalData[current];
}

```

Note that in this example, the parameter `devNum` is ignored. This is not always the case. Indeed, this button driver would likely have support for more than one button, and in that case, we would use `devNum` as the index into an array or vector containing data sampled from all the buttons.

There are other methods that must be implemented depending on the classes from which a given driver class derives. In the joystick example given earlier, the method `getAnalogData()` would have to be implemented in addition to `getDigitalData()`. The prototype for `getAnalogData()` is:

```

virtual float getAnalogData(int devNum = 0)

```

The joystick driver would use this to return values for the X and Y axes. The data here is more complex because it would be for triple-buffered two-dimensional samples. An implementation might look similar to the following:

```

float MyNewJoystickDevice::getAnalogData(int axis)
{

```

```
    vjASSERT(axis >= 0 && axis <= 1 && "only 2 axes (x and y) available");  
    return mSampledAnalogData[current][axis];  
}
```

In this driver, the integer argument to the method is used to represent either the X or the Y axis. The assertion ensures that a valid axis index is passed.

Register the Device Driver with VR Juggler

Device driver registration is done through a templated constructor called `vjDeviceConstructor`. It is used as follows:

```
#include <Input/InputManager/vjDeviceFactory.h>  
  
vjDeviceConstructor<MyNewButtonDevice>* this_ptr_not_used =  
    new vjDeviceConstructor<MyNewButtonDevice>;
```

The new device driver can be compiled along with the `vjDeviceConstructor` call into a standalone library (`.a`, `.so`, `.lib`, and `.dll` are common). This library can then be linked with applications. In this way, there is no need to modify the VR Juggler source code to add a new driver. The driver code can be centralized into a single, neat driver that can be distributed as a “plug-in” for VR Juggler.

Note

When linking, it is important that the linker include *all* library symbols. On IRIX with the MIPSpro Com#pilers, the `-all` option informs the linker of this need. The GNU linker uses `--whole-archive`. Refer to your compiler documentation for more detailed information.

Device Driver Configuration

To configure a device, two things are needed:

1. Configuration files
2. Driver code that accepts the configuration

Configuration Files

Before configuring a device, a new configuration chunk description must be created. We recommend that this be done using `VjControl`. For the button device, the chunk description will look similar to the following:

```
chunk MyButtonDevice "MyButtonDevice" "Configuration for my one-button device"  
  Name String 1 "Name" "Unique name of an instance of this chunk type"  
  port String 1 "Port" "Serial port this device is connected to"  
  baud Int 1 "Baud Rate" "Serial port speed"  
end
```

For a real device, all of these parameters may or may not be needed. (In VR Juggler 1.0, `vjInput` requires `baud` and `port`, but this has been corrected in newer versions.) Again, the `VjControl` chunk description editor simplifies the creation of this description so that only the required elements are present.

Note

If the new driver is for a positional device, its name must be added to the enumeration in the `PosProxy` chunk description. `VjControl` aids this addition.

Once the chunk description is in place, a new configuration chunk can be created. Once again, `VjControl` makes the step easier. The following is an example configuration file that configures the one-button device we have been using thus far:

```
vjincludedescfile
  Name "mybuttondevice.desc"
end
MyButtonDevice
  Name "Button Device"
  port { "/dev/ttyd4" }
  baud { "9600" }
end
End
```

Writing Code that Accepts the Configuration

In the driver, there are two methods that must be implemented in order to handle config chunks:

1.

```
static std::string MyNewButtonDevice::getChunkType();
```

When the VR system configuration changes, the system asks every registered driver for their chunk type. If it matches the new configuration chunk passed into the system, then the driver's `config()` method is invoked. This will happen, for example, when a new configuration file is loaded into the VR Juggler kernel. The implementation of this method was described above.

2.

```
virtual bool VjInput::config(VjConfigChunk* c);
```

When the system detects a configuration change for a given driver, it will pass the new `VjConfigChunk` object to this method. For more information about how to access config chunk objects, refer to the *Programmer's Reference*. The following is a simple example for the basic button device we have used thus far:

```
bool MyNewButtonDevice::config( VjConfigChunk *c )
{
  if (!VjDigital::config(c))
    return false;

  port_id = c->getProperty("port");
  baudRate = c->getProperty("baud");

  return true;
}
```

Example Code

Now that we have explained the concepts involved in adding a device driver to VR Juggler, we can show some code. The following example is for a fictitious piece of hardware that has only one button.

```
1 #include <Input/vjInput/vjDigital.h>
  #include <Input/InputManager/vjDeviceFactory.h>
```

```

#include <Threads/vjThread.h>
#include <Sync/vjGuard.h>
5

class MyButtonDevice : public vjDigital
{
public:
10   MyButtonDevice() : mSampleThread(NULL)
      {;}

      virtual ~MyButtonDevice()
15     {
          this->stopSampling();
      }

      virtual void  getData();
      virtual int   startSampling();
20     virtual int   sample();
      virtual int   stopSampling();
      static std::string getChunkType();

private:
25     static void   threadedSampleFunction(void* classPointer);
      int           mDigitalData;
      vjThread*     mSampleThread;

      // configuration data set by config()
30     int           mPortId, mBaud;
};

vjDeviceConstructor<MyButtonDevice>* this_ptr_not_used = new vjDeviceConstructor;

35 //: What is the name of this device?
// This function returns a string that should match this device's
// config chunk name.
static std::string MyButtonDevice::getChunkType()
{
40   return std::string("MyButtonDevice");
}

// spawn a sample thread,
// which calls MyButtonDevice::sample() repeatedly
45 int MyButtonDevice::startSampling()
{
    mSampleThread = new vjThread(threadedSampleFunction, (void*)this);

    if ( !mSampleThread->valid() )
50     return 0; // thread creation failed
    else
        return 1; // thread creation success
}

55 //: Record (or sample) the current data
// this is called repeatedly by the sample thread created by startSampling()
int MyButtonDevice::sample()
{
    // here you would add your code to
    // sample the hardware for a button press:
60     mDigitalData[progress] = rand_number_0_or_1();
    return 0;
}

65 // kill sample thread
int MyButtonDevice::stopSampling()

```

```
{
    if ( mSampleThread != NULL )
    {
70         mSampleThread->kill();
            delete mSampleThread;
            mSampleThread = NULL;
        }
    return 1;
75 }

//: function for users to get the digital data.
// here we overload vjDigital::getDigitalData
int MyButtonDevice::getDigitalData(int d)
80 {
    // only one button, so we ignore "d"
    return mDigitalData[current];
}

85 // Our threaded sample function
// This function is declared as a static member of MyButtonDevice
// just spins... calling sample() over and over.
void MyButtonDevice::threadedSampleFunction(void* classPointer)
{
90     MyButtonDevice* this_ptr = static_cast<MyButtonDevice*>( classPointer );

        // spin until someone kills "mSampleThread"
        while (1)
        {
95             this_ptr->sample();
                sleep(1); //specify some time here, so you don't waste CPU cycles
        }
    }

100 //: When the system detects a configuration change for your driver, it will
// pass the new vjConfigChunk into this function. See the documentation
// on config chunks, for information on how to access them.
bool MyButtonDevice::config(vjConfigChunk *c)
{
105     if ( !vjDigital::config(c) )
        return false;

        mPortId = c->getProperty("port");
        mBaud = c->getProperty("baud");
110     return true;
    }
}
```

Index

A

- application object, 6, 6, 6
 - base interface of, 9
 - benefits of, 6
 - frame functions, 10
 - initialization, 9
 - overview, 6
- application programming
 - getting input, 38
 - OpenGL, 41
 - context-specific data, 43
 - drawing, 42
 - OpenGL Performer, 49
 - scenegraph access, 49
 - scenegraph initialization, 49
 - VTK, 53
 - where to get device input, 39
- applications
 - application object overview, 6
 - basics, 6
 - device access, 32
 - main function
 - structure, 7
 - use, 7
 - starting, 7
 - writing, 38

C

- CAVELib
 - porting to VR Juggler, 54
- classes
 - vjAnalog, 82
 - vjApp, 6, 6, 9, 11, 11, 38, 38, 41
 - vjConfigChunkHandler, 79
 - vjDeviceConstructor, 85
 - vjDeviceInterface, 32, 39
 - (see also vjDeviceInterface)
 - vjDigital, 82
 - vjDigitalInterface, 32, 34
 - vjDigitalProxy, 34
 - vjGLApp, 11, 38, 41, 42
 - (see also vjGLApp)
 - vjGlove, 82
 - vjGloveGesture, 82
 - vjInput, 82
 - vjKeyboardInterface, 32
 - vjMatrix, 20
 - (see also vjMatrix)
 - vjPfApp, 11, 49
 - (see also vjPfApp)
 - vjPosInterface, 32
 - vjPosition, 82, 82
 - vjProxy, 34

- (see also vjProxy)
- vjSemaphore, 76
- vjSimAnalog, 82
- vjSimDigital, 82
- vjSimDigitalGlove, 82
- vjSimPosition, 82
- vjThread, 67, 72, 73, 74
- vjVarValue, 81
- vjVec3, 12, 13
 - (see also vjVec3)
- vjVec4, 12, 13
 - (see also vjVec4)
- context-specific data, 43
 - details, 47
 - use of, 45
- context-specific variables, 45

D

- device aliases
 - examples of, 33
- device drivers
 - adding, 82
 - configuring, 85
 - example, 86
 - guide, 82
 - implementing, 82
 - registering, 85
- device interfaces
 - as smart pointers
 - (see also smart pointer)
 - initialization of, 33
- device proxies
 - access through device interfaces
 - (see also vjDeviceInterface)
- Draw Manager, 6
 - application classes, 11
 - OpenGL, 11
 - OpenGL Performer, 11

F

- frame, 8
 - definition of, 8
- frame of execution, 8
- functor, 68, 72
 - use with vjThread
 - (see also vjThread)

G

- GLUT
 - porting to VR Juggler, 59

I

- input device types, 38

M

- main function, 6

multi-threading, 67
 helper classes, 67
 techniques, 67
 vjThread, 67

N

NSPR, 72

O

OpenGL rendering contexts, 44

P

pfMatrix
 converting from vjMatrix
 (see also vjMatrix)
pfVec3
 converting from vjVec3
 (see also vjVec3)
porting applications from
 CAVElib, 54
 GLUT, 59
proxy
 application-level access, 32
 definition of, 32

R

run-time reconfiguration, 79
 use in application, 79

S

smart pointer, 32

T

thread of control, 67
tutorial
 drawing a cube using display lists, 47
 drawing a cube with OpenGL, 42
 getting input, 40
 loading a model with OpenGL Performer, 49

V

virtual platform, 6
vjApp
 configAdd() method, 80
 configCanHandle() method, 79
 configRemove() method, 80
vjBaseThreadFunctor
 (see also vjThreadMemberFunctor, vjThreadNon#
 MemberFunctor)
 description of, 72
 details, 74
 setArg() method, 74
vjConfigChunk
 getNum() method, 81
 getProperty() method, 80
 getType() method, 80

vjDeviceInterface, 32
 description of, 32
 details, 33
 subclasses of, 33
vjGLApp
 contextInit() method, 46
 contextPreDraw() method, 46
 draw() method, 42
 extensions to vjApp, 41
vjMatrix, 18
 adding, 22
 assigning, 21
 compared to C++ matrices, 19
 constraining rotation, 26
 converting to pfMatrix, 30
 creating, 20
 description of, 19
 details, 31
 equality comparison, 21
 extracting transformation information, 30
 inverting, 22
 making
 Euler rotation, 25
 from quaternion, 26
 identity, 25
 rotation about a single axis, 27
 scale transformation, 29
 translation transformation, 28
 using direction cosines, 26
 multiplying, 23
 scaling, 24
 subtracting, 23
 transposing, 22
 using with OpenGL, 31
 zeroing, 25
vjMutex, 76
 creating, 76
 description of, 76
 details, 77
 locking, 77
 without blocking, 77
 releasing, 77
 testing state, 77
vjPfApp
 appChanFunc() method, 51
 configPWin() method, 52
 drawChan() method, 52
 extensions to vjApp, 49
 getFrameBufferAttrs() method, 52
 getScene() method, 49
 initScene() method, 49
 postDrawChan() method, 52
 preDrawChan() method, 52
 preForkInit() method, 51
vjProxy, 34
 as pointer to physical device, 34
 description of, 34
 details, 34

-
- getData() method, 34
 - vjSemaphore, 75
 - creating, 75
 - description of, 75
 - details, 76
 - locking, 75
 - releasing, 75
 - reset() method, 75
 - vjThread, 67
 - canceling, 71
 - creating, 68
 - description of, 68
 - details, 71
 - joining threads, 69
 - priority, 70
 - self, 71
 - sending signals, 71
 - suspend, resume, 70
 - use, 68
 - vjThreadMemberFunctor
 - details, 74
 - use, 73
 - vjThreadNonMemberFunctor
 - details, 74
 - use, 74
 - vjVec3, 12
 - adding, 17
 - assigning, 16
 - converting to pfVec3, 15
 - creating, 13
 - cross product, 17
 - description of, 12
 - details, 18
 - dividing by a scalar, 15
 - dot product, 16
 - equality comparison, 16
 - inverting, 13
 - length of, 14
 - multiplying by a scalar, 14
 - normalizing, 14
 - subtracting, 17
 - transforming by a matrix
 - full, 17
 - partial, 18
 - vjVec4, 12
 - adding, 17
 - assigning, 16
 - creating, 13
 - description of, 12
 - details, 18
 - dividing by a scalar, 15
 - dot product, 16
 - equality comparison, 16
 - inverting, 13
 - length of, 14
 - multiplying by a scalar, 14
 - normalizing, 14
 - subtracting, 17
 - transforming by a matrix
 - full, 17
 - partial, 18
-